Résolution des interférences pour la composition dynamique de services en informatique ambiante

Sana Fathallah Ben Abdenneji

UNIVERSITÉ NICE SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université Nice Sophia Antipolis
Mention : INFORMATIQUE

Présentée et soutenue par

Sana FATHALLAH BEN ABDENNEJI

Résolution des interférences pour la composition dynamique de services en informatique ambiante

Thèse dirigée par Michel RIVEILL

préparée au sein du laboratoire I3S, Equipe RAINBOW

soutenue le 19-12-2013

Jury :

Rapporteurs :	Lionel SEINTURIER	-	Professeur,Université Lille1
	Mokrane BOUZEGHOUB	-	Professeur, Université Versailles SQY
Directeur :	Michel RIVEILL	-	Professeur, Université Nice Sophia Antipolis
Co-Directeur :	Stéphane LAVIROTTE	-	Maître de conférence, Université Nice Sophia Antipolis
Président :	KamelHAMROUNI	-	Professeur, Université Tunis el Manar
Examinateurs :	Georges DA COSTA	-	Maître de conférence, Université Toulouse III
Invité :	Jean-Yves TIGLI	-	Maître de conférence, Université Nice Sophia Antipolis

Remerciements

Je tiens en tout premier lieu à remercier Stéphane Lavirotte pour son encadrement durant cette thèse. J'ai énormément apprécié votre encadrement pendant ces années et j'ai pris beaucoup de plaisir à travailler avec vous. Votre grand recul et votre cadrage m'ont permis de canaliser mes travaux de recherche, tout en me laissant une grande marge d'autonomie. Vous avez su à chaque moment me donner les bons conseils et orientations lorsque j'étais en proie au doute ou à l'hésitation. Je remercie également Michel Riveill d'avoir suivi et dirigé cette thèse. Merci d'avoir été présent quand j'avais besoin. Je remercie également tout particulièrement Jean-Yves Tigli pour m'avoir suivi, encouragé et aidé tout au long de cette thèse, et ce, toujours dans une ambiance formidable. Je souhaite remercier Gaëtan Rey pour son soutien et ses remarques.

Mes remerciements d'adressent également aux membres de jury : les deux rapporteurs de cette thèse Lionel Seinturier et Mokrane Bouzeghoub, l'examinateur Georges Da Costa, et le président de ce jury Kamel Hamrouni.

Je tiens aussi à remercier Daniel Gaffé et Annie Ressouche pour leur collaboration sur les automates présentés dans ce manuscrit. Je garde un très bon souvenir de cette collaboration.

Je souhaite remercier les thésards du pôle GLC avec qui j'ai passé de bons moments : Christophe Vergoni, Christion Brel, Amel Ben Othmane, Macha Da Costa, Marwa Hassen, Simon Urli, Andrés Moreno. Je souhaite bonne chance à ceux qui doivent soutenir leur thèse dans les années à venir.

Enfin, je tiens à remercier ma famille. Mehdi, merci pour ta présence, ton soutien, tes conseils, ton amour et d'avoir supporté mon stress. Mes parents, merci pour votre soutien inconditionnel et permanent.

Résumé

Comme dans de nombreux autres domaines, la construction des applications en Informatique Ambiantes (IAm) se fait par réutilisation d'entités logicielles disponibles. Pour des raisons de conductivités, de pannes, de charge de batterie mais aussi de nombreuses autres, la disponibilité de ces entités est imprévisible ce qui implique que l'auto-adaptation dynamique des applications est une nécessité. Cela passe par la spécification en parallèle des adaptations par des experts de divers domaines. Ce parallélisme de construction, peut amener des problèmes d'interférences lors de la composition dynamique de plusieurs adaptations.

Dans cette thèse, par l'utilisation de graphes, nous contribuons à la définition d'un cadre formel pour la détection et la résolution de ces interférences. L'assemblage des entités logicielles repose sur des connecteurs d'assemblage qui sont utilisés dans la spécification des adaptations. Des règles de réécriture de graphe permettront de résoudre les interférences détectées, cette résolution étant guidée par la connaissance de connecteurs définis. De plus, pour pouvoir étendre dynamiquement et automatiquement notre mécanisme de gestion des interférences, nous proposons la modélisation comportementale de ces connecteurs. Ceci permet de ne pas reposer sur une connaissance à priori des connecteurs et autorise par la même d'étendre dynamiquement l'ensemble des connecteurs disponibles pour la spécification des adaptations.

Abstract

Like many other fields, application construction in ubiquitous computing is done by reuse of available software entities. For reasons of conductivity, breakdown, battery charge, but also many others reasons, the availability of these entities is unpredictable. As consequence, the self-adaptation of applications becomes necessary. This requires the specification of parallel adaptations by experts from various fields. This parallel specification can cause interference problems when several adaptations are composed.

In this thesis, using graph formalism, we contribute to the definition of a formal approach for the detection and the resolution of interferences. The specification of adaptation uses connectors in order to assemble software entities. Graph rewriting rules are defined to solve the detected interferences. This resolution is guided by the knowledge of defined connectors. In addition, in order to extend dynamically and automatically our interference management mechanism, we propose behavioral modeling of these connectors. This allows us extending our mechanism without an a priori knowledge of connectors and allows afterwards to extend the set of available connectors used for adaptations' specifications.

Table des matières

Première partie

Introduction et Analyse de l'État de l'art

CHAPITRE 1

Introduction

Sommaire

1.1 L'informatique ambiante

1.1.1 Présentation

L'informatique ambiante (appelée aussi *ubiquitous computing*[Wei93], *pervasive computing*[Sat01]) a été proposée par Mark Weiser [Wei91] comme une vision de l'évolution de l'informatique. Cette nouvelle forme de l'informatique vise à assister implicitement et discrètement un utilisateur dans les tâches qu'il accomplit au quotidien. Cette vision révolutionnaire a inspiré de nombreux travaux dans divers domaines tel que les système embarqués, les communications sans fil, etc. Les évolutions réalisées dans les systèmes embarqués ont enrichi la vision de Weiser en proposant des systèmes informatisés ayant des tailles de plus en plus petites et intégrés dans les objets de la vie quotidienne. Nous appelons ces derniers des *dispositifs* (par exemple un capteur de température, une lampe communicante, etc.). Parallèlement à cela, l'apparition de standards de télécommunication pour ces dispositifs, tels que WiFi, Bluetooth et ZigBee, a permis à ces équipements de communiquer ensemble pour coopérer au sein d'un environnement ambiant. Tout cela se fait actuellement en interaction avec l'utilisateur, mais cela doit évoluer vers des communications entre les dispositifs de manière transparente pour l'utilisateur en minimisant au maximum son intervention explicite.

Satyanarayanan [Sat01] a proposé le concept de *distraction minimale* comme approximation de l'invisibilité introduite par Mark Weiser. L'idée est de faire intervenir l'utilisateur le moins possible et d'essayer d'anticiper les besoins. Cela permet aux utilisateurs d'interagir avec les systèmes ambiants au niveau du subconscient, c'est à dire qu'ils y prêteront attention seulement quand il y a un besoin à satisfaire non pris en compte. Cette nouvelle approche de l'informatique s'intègre alors à la mobilité de l'utilisateur qui devient capable d'accéder aux services n'importe où et à n'importe quel moment (*anytime anywhere*).

La définition des applications destinées à fonctionner dans ce type d'environnement fait apparaître de nouveaux challenges et implique la prise en considération de diverses contraintes. La section suivante présentera les contraintes auxquelles nos travaux sont confrontés.

1.1.2 Contraintes

Un système ambiant repose principalement sur un ensemble de dispositifs physiques qui interagissent les uns avec les autres. À cela s'ajoute des entités logicielles qui ont été conçues pour fournir des services à l'utilisateur. Nous appelons cet ensemble de dispositifs et entités logicielles qui composent un système *l'infrastructure logicielle*. La conception d'un système ambiant est alors axée sur les propriétés associées à cette infrastructure. En fait, toute application destinée à être exécutée dans un environnement ambiant doit respecter les contraintes imposées par ce cadre de travail.

1.1.2.1 Multiplicité des entités (ε_1)

Un système ambiant implique des entités multiples et diverses dont le nombre n'est pas fixé à priori (non borné). Ces entités peuvent prendre plusieurs formes et être physiques (imprimante, téléphone, réfrigérateur, etc.) ou logicielles (système de réservation, etc.). En effet, tout au long du cycle de vie d'une application, des évolutions peuvent affecter l'ensemble des entités qui la composent. Un système ambiant doit donc considérer un grand nombre d'entités et de nombreuses variantes qui peuvent se trouver dans son environnement.

1.1.2.2 Briques de base boîte noire (ε_2)

Les dispositifs ainsi que les entités logicielles fournis par des constructeurs ne sont pas prévus pour être modifiés : ce sont des boîtes noires. Ce concept limite les interactions possible à l'utilisation des services qu'ils fournissent et empêche d'accéder directement à leur implémentation et donc de la modifier ou de l'adapter en interne. La création d'un système ambiant ne peut donc en aucun cas passer par une modification du comportement interne de ces entités mais simplement favoriser le principe de la réutilisabilité, puisqu'on choisit une entité pour sa fonctionnalité et non son implémentation.

1.1.2.3 Imprévisibilité de disponibilité (ε_3)

Dans la vision de l'informatique ambiante, l'utilisateur et les dispositifs évoluent dans un environnement variable et potentiellement imprévisible dans lequel les entités impliquées apparaissent et disparaissent opportunément (une conséquence de la mobilité, des déconnexions, pannes, etc.). Dès lors, l'hypothèse implicite d'avoir un ensemble d'entités connues à la conception du système ne tient plus en IAm. Il n'est pas possible d'anticiper à la conception quels seront tous les dispositifs qui vont être disponibles et à quel moment. En rupture avec les systèmes rigides modelés sur mesure pour un usage donné, nous devons donc explorer la construction des applications avec comme contrainte l'imprévisibilité de la disponibilité des entités qui la composent et une connaissance partielle de celles-ci.

1.1.2.4 Multi Designer et Indépendance (ε_4)

La multitude d'entités impliquées dans un espace ambiant ainsi que la diversité de préoccupation que nous pourrons avoir, rend non envisageable d'avoir un seul développeur qui détient la maîtrise totale de ces entités pour la spécification complète du système. Le système sera spécifié par divers designers dont chacun est expert dans un domaine particulier. Les designers ou concepteurs ne sont

pas censés connaître la présence des autres parties qui forment le système (spécifiés par les autres designers). Par conséquence, il y aura une *indépendance* de la spécification des sous-parties du système.

Nous pouvons constater que, pour être mise en œuvre, les contraintes présentées ci-dessus nécessitent l'adaptation de la structure de l'application et/ou des entités logicielles qui la composent. La caractéristique principale d'une application en IAm est donc son *adaptabilité* à différents éléments de son environnement d'exécution (nous pouvons l'assimiler au *contexte*).

1.2 Besoin d'adaptation sensible au contexte

1.2.1 Définitions

1.2.1.1 Sensibilité au contexte

Du fait de la multitude des entités impliquées dans le système (ε_1), de leur imprévisibilité de disponibilité (ε_3) et de l'évolution dynamique des besoins des utilisateurs, les applications doivent gérer ce dynamisme et s'adapter à de nouvelles configurations non prévues à la conception [MSKC04]. Adapter un système logiciel [MK96] signifie modifier le système afin de lui permettre de se comporter correctement dans des contextes différents pour garantir la disponibilité des services proposés. La nécessité de répondre à une situation non prévue lors de la phase de conception impose à l'application d'être sensible au contexte.

Au regard de la littérature, il apparaît qu'il existe plusieurs définitions pour la notion de contexte. Nous considérons la définitions de *Chaari et al* [CLF05] : *"Le contexte est l'ensemble des paramètres externes à l'application qui peuvent influer sur son comportement en définissant de nouvelles vues sur ses données et ses fonctionnalités. Ces paramètres ont un aspect dynamique qui leur permet d'évoluer durant le temps d'exécution"*. Un système est dit *sensible au contexte* s'il est capable d'utiliser les informations contenues dans le contexte dans le but de s'adapter. Dey [Dey00], l'un des premiers chercheurs dans le domaine de la sensibilité au contexte, a identifié trois étapes nécessaires pour qu'une application soit sensible au contexte. En premier lieu, il est nécessaire d'avoir des mécanismes pour détecter les changements de contexte de l'application. Un tel système doit être capable de *percevoir* à un instant donné tous les éléments de son contexte. Ensuite, il faudra effectuer une interprétation du contexte. Cette interprétation sera exploitée par un mécanisme de décision pour définir l'ensemble des modifications à apporter à l'application. Le mécanisme de décision fait alors la mise en correspondance entre la perception du contexte et l'adaptation souhaitable. La capacité d'une application sensible au contexte à interpréter son contexte et à produire automatiquement l'ensemble des modifications à effectuer servira à concevoir des mécanismes d'auto-adaptation.

1.2.1.2 L'auto-adaptation

Un système est qualifié d'***auto-adaptatif*** lorsque l'adaptation est déclenchée par des variations du contexte d'exécution et qu'elle est réalisée de façon automatique par le système lui-même sans l'intervention de l'utilisateur. Dans [OGT+99], l'auto-adaptation est définie comme suit : *"Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation"*. Pour ce faire, il faudra que le système

ait la capacité d'agir sur lui-même : c'est l'*intercession*. Un mécanisme d'auto-adaptation respecte donc la vision de Mark Weiser par rapport de l'invisibilité (assister implicitement et discrètement un utilisateur sans le gêner dans l'accomplissement de ses tâches principales). Pour simplifier, par la suite, nous utilisons le terme adaptation pour désigner l'adaptation automatique ou auto-adaptation.

Un mécanisme d'adaptation possède des propriétés liées au cadre dans lequel il va être appliqué. Dans la section suivante, nous nous intéressons aux propriétés que doivent avoir les mécanismes d'adaptation par rapport à notre cadre de travail. Nous avons choisi de détailler les critères qui ont une relation avec la problématique qui va être traitée dans cette thèse.

1.2.2 Des critères menant vers le problème d'interférence

Salehie et Tahvildari [ST09] ont proposé dans leur travaux une taxonomie qui classifie les propriétés de l'adaptation. On y trouve notamment des éléments qui spécifient *"comment réaliser l'adaptation ?"* et *"quoi adapter ?"*. Les propriétés ont été décrites de manière générale en spécifiant les différentes alternatives pour chaque propriété indépendamment de tout domaine d'application. Nous reprenons quelques propriétés mais cette fois-ci en effectuant les choix qui répondent aux contraintes présentées précédemment.

1.2.2.1 Adaptation externe

L'adaptation interne inclut le code de l'adaptation dans l'application. Le problème de cette technique est que la modification du comportement lié à l'adaptabilité nécessite la modification de l'application, ce qui complexifie cette tâche. Cette technique est à employer dans le cas d'applications dont le comportement dynamique est limité et standardisé. La multiplicité d'entités et leur imprévisibilité de disponibilité implique un comportement dynamique qui n'est pas borné. En IAm, l'adaptation doit donc être **externe** (σ_1) ce qui permettra de faire évoluer la stratégie d'adaptation indépendamment de l'application. On suit alors la logique de *séparation des préoccupations* [HL95]. Il existe une entité, souvent appelée gestionnaire d'adaptation, qui surveille le système et commande les changements. L'avantage de cette technique est que le code d'adaptation de l'application est libéré de toutes contraintes liées à l'application et pourra être personnalisé et configuré pour différents systèmes.

1.2.2.2 Adaptation ouverte

Un mécanisme d'adaptation fermé dispose d'un nombre fixe de possibilités d'adaptation. Il n'est pas possible d'introduire de nouveaux comportements au cours de l'exécution en dehors d'un ensemble prédéfini. Dans une adaptation ouverte, le mécanisme d'adaptation peut être étendu et, par conséquent, de nouvelles alternatives d'adaptations peuvent être ajoutées. Le problème de l'adaptation fermée est que, pour satisfaire les besoins actuels, il faudra avoir anticipé tout nouveau cas et définir l'adaptation qui lui correspond. L'imprévisibilité et la multitude d'entités impliquées dans un système ambiant rend la définition d'un ensemble d'adaptations défini statiquement très difficile car l'ensemble des contextes peut être infini. Il n'est pas non plus possible de connaître à priori toutes les variations du contexte. Pour toutes ces raisons, en IAm il faudra que le mécanisme d'adaptation soit **ouvert** (σ_2) et devra permettre **l'extension de l'ensemble des adaptations**.

1.2.2.3 Adaptation compositionnelle

Une question primordiale pour un mécanisme d'adaptation est : *quelles parties de l'application peuvent être sujet à un changement* ? Une adaptation peut modifier les modules ou l'architecture et la façon dont ils sont composés. Deux types d'approches d'adaptation sont donc possibles : *orientées boîtes blanches* ou *orientées boîtes noires*. Les techniques d'adaptation orientées boîtes blanches exigent la connaissance de l'implémentation interne des entités à adapter et de l'ensemble des modifications que nous voulons apporter. Cependant, parce que l'on ne peut pas imaginer qu'il soit possible d'avoir le détail d'implémentation des dispositifs et des entités logicielles du système, une telle technique est difficile à concevoir pour notre cadre d'étude. En effet, comme nous avons pu le voir, les entités logicielles sont des boîtes noires (ε_2). De ce fait, la technique d'adaptation doit considérer cette contrainte et proposer une technique orientée boîte noire. Cette adaptation vise à intégrer de nouvelles entités qui n'avait pas été prévues à la conception, supprimer des entités et échanger des entités par d'autres. Il s'agit de l'**adaptation compositionnelle** (σ_3) tel que définie dans [MSKC04] et [Fer11]. Il a été démontré dans la littérature [GRWK09] [AZI+08] que ce type d'adaptation est particulièrement approprié à la gestion de la variation de l'infrastructure logicielle d'une application qui caractérise le cadre de l'IAm.

Synthèse

La *séparation des préoccupations* d'adaptation donne naissance à des *entités d'adaptation* (σ_1). Cela permettra d'ajuster la stratégie d'adaptation d'une manière externe au mécanisme d'adaptation. Ces entités d'adaptation seront spécifiées par divers designers dont chacun détient seulement

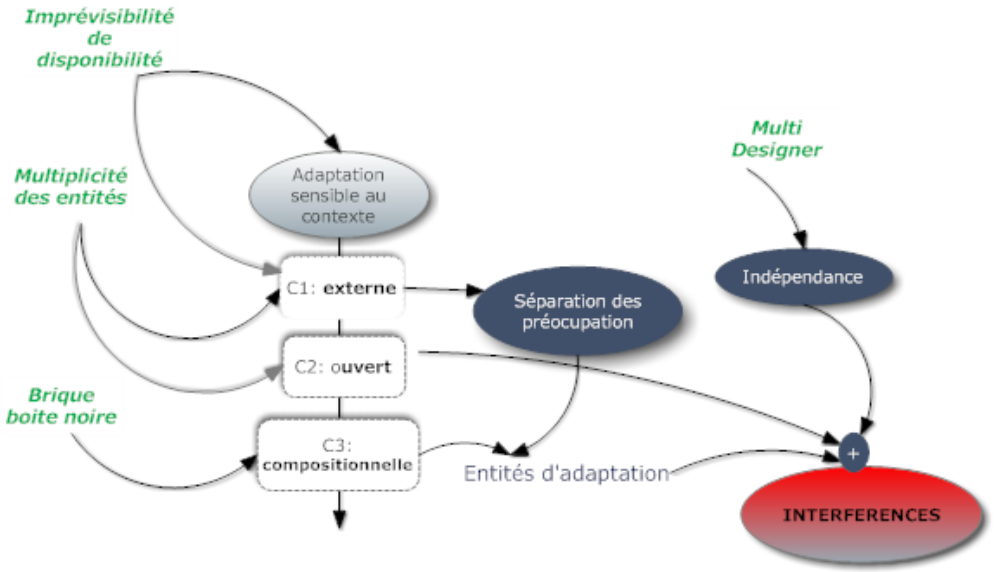


FIGURE 1.1 – Caractéristiques de l'IAm et critères du mécanisme d'adaptation

la connaissance des parties du système qui le concerne (ε_4). De ce fait, les entités d'adaptation sont spécifiées *indépendamment* les unes des autres.

Cette indépendance à la conception n'est pas vraiment conservée au moment où ces entités vont être intégrées dans un même système. De ce fait, des interactions pourront apparaître suite à leur

composition. Dans le cas où les interactions modifient le comportement attendu par au moins une de ces entités d'adaptation, on parle d'un problème d'**interférence**. C'est la problématique que nous adressons dans cette thèse. Nous cherchons à définir un mécanisme de gestion des interférences adapté à notre contexte de travail. La résolution de ces problèmes d'interférence ne pourra pas être faite au cas par cas car l'ensemble des entités d'adaptation est non borné (adaptation ouverte (σ_2)). La gestion des interférences entre les adaptations devra alors être déléguée au mécanisme d'adaptation. La figure 1.1 illustre le lien entre les contraintes et le problème d'interférence.

1.3 Enjeux et Objectifs

Pour adapter les applications sensibles au contexte, nous avons vu que l'adaptation compositionnelle (σ_3) permet de prendre en compte dynamiquement les parties logicielles exposées par des dispositifs qui viennent d'apparaître et inversement lors de leur disparition. Dans cette section, nous nous intéressons tout d'abord aux techniques permettant la spécification des entités qui constituent le système. Ces entités sont ensuite utilisées pour la spécification des adaptations à faire. Ces entités d'adaptation devront être spécifiées en parallèle par divers développeurs spécialistes de leur domaine. Nous verrons que la programmation orientée aspect répondra à ces besoins. Par la suite, les entités d'adaptation sélectionnées pour construire le système (ou bien le faire évoluer) peuvent être en interférence lors de leur intégration. Nous définirons alors les problèmes d'interférence auxquels nous nous intéressons dans ce manuscrit. Enfin, à partir de l'ensemble des caractéristiques, défis et questions que nous venons de soulever, nous définirons les objectifs de cette thèse.

1.3.1 Entités logicielles

Pour pouvoir réaliser une adaptation compositionnelle, le système doit être construit d'une manière modulaire avec un couplage faible entre les entités logicielles qui le composent. Plusieurs techniques offrent une modularité mais peu garantissent le couplage faible. Le découpage architectural peut être obtenu en concevant les applications à partir de paradigmes de programmation spécifiques : la programmation orientée composant et la programmation orientée service.

La programmation orientée composant La programmation orientée composant peut être considérée comme le successeur de l'approche orientée objet. L'unité élémentaire qui forme une application est le composant qui peut être vu comme un regroupement d'objets. Ce modèle de programmation propose une meilleure encapsulation. La définition du composant la plus citée est celle de C. Szyperski [SGM02] :

> **Définition 1 :**
> *A software component is a unit of composition which contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Un composant peut être décrit comme étant une entité logicielle préfabriquée qui est conçue pour être composée avec d'autres composants. Un type de composant est une entité logicielle abstraite qui se caractérise par des interfaces qui peuvent être de deux types : (1) requises (sorties) ou (2) fournies (entrées). Les interfaces requises permettent de définir les dépendances entre composants. Les interfaces fournies définissent les capacités offertes par un composant. Une application prend donc la forme d'un ensemble d'instances de composants appelés assemblage de composants.

La programmation orientée service Il existe plusieurs définitions du paradigme de service. On considère la définition d'OASIS (Organization for the Advancement of Structured Information Standards) [HJ06] :

> **Définition 2 :**
> *A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface [...]. A service is opaque in that its implementation is typically hidden from the service consumer [...].*

Cette définition montre qu'un service est forcément une entité d'encapsulation boîte noire où seules les informations nécessaires pour sa sélection et son invocation sont disponibles. La façon dont il est implémenté est inconnue et ne peut pas être modifiée, ce qui correspond aussi au fonctionnement des dispositifs. Le service est un mécanisme permettant d'accéder à une ou plusieurs fonctionnalités uniquement via des points d'accès qui sont les interfaces du service.

Les frontières entre les paradigmes composant et service sont assez floues puisqu'ils offrent des fonctionnalités très proches, si ce n'est similaires. Les applications peuvent donc se baser sur des entités logicielles qui peuvent être composant et/ou service. A partir de ce constat, nous proposons *un mécanisme de gestion des interférences qui n'est pas lié aux technologies d'implémentation*. Que ce soit service ou bien composant, notre mécanisme de gestion des interférences les considère tous comme des entités logicielles réutilisées pour la construction du système. De ce fait, nous cherchons à définir un mécanisme de gestion des interférences qui pourra être intégré dans des approches orientées composant ou service.

Du fait de ne pas distinguer les paradigmes service et composant et de les considérer comme des entités logicielles, la spécification de composition doit être indépendante d'une technologie donnée (puisqu'elle pourra être utilisée pour définir une composition de service ou un assemblage de composants). Il est donc indispensable que celle-ci soit réalisée à un niveau élevé d'abstraction pour permettre aux développeurs de se concentrer sur la logique métier à mettre en œuvre par la composition sans se préoccuper de leur technologie d'implémentation.

1.3.2 Les entités d'adaptation

Dans notre cadre de travail, l'environnement est caractérisé par l'imprévisibilité et la dynamicité. Il contient des entités diverses que nous ne pouvons pas connaître a priori. Pour toutes ces raisons, nous ne pouvons pas détenir la connaissance de la composition complète du système que nous voulons construire, ni même l'ensemble des entités qui seront présentes à un moment donnée. Au lieu de spécifier une unique entité qui décrit tout le système, il est donc nécessaire de spécifier des fragments de composition ou entités d'adaptation dont chacune exprime une préoccupation. Une entité d'adaptation abstraite indique les entités logicielles abstraites (au niveau type et pas instance) impliqué dans la composition ainsi que leur interaction. Le passage d'une entité d'adaptation abstraite vers une concrète est déterminé au moment de l'exécution selon la disponibilité des entités logicielles de l'environnement. Les entités d'adaptation vont alors être spécifiées par plusieurs développeurs dont chacun possède une expertise particulière. *"Comment définir alors les entités d'adaptation avec la minimisation du nombre d'entités à écrire ?"* et *"Sur quel approche les entités d'adaptation peuvent reposer ?"*. Nous avons donc besoin d'un mécanisme qui nous permette de définir ces entités d'adaptation d'une manière indépendante les unes des autres et de manière à pouvoir les composer

facilement pour construire le système final.

Un mécanisme, très utilisé dans la littérature, permettant une séparation explicite des préoccupations ainsi qu'une spécification des fonctionnalités diverses et indépendantes est l'*AOP* (Aspect Oriented Programming ou Programmation Orientée Aspect). Nous présentons dans la suite ce concept de programmation.

1.3.2.1 AOP

La conception d'un système logiciel est basé sur la séparation des préoccupations de base (basic concerns) en les encapsulant dans des entités réutilisables. L'hypothèse de travail est que les mécanismes de modularité sont supportés jusqu'à maintenant par l'unique décomposition hiérarchique logicielle basée, par exemple, sur les fonctionnalités offertes. Malgré cette séparation, il existe toujours un entrelacement (figure 1.2.a) entre le code métier et les préoccupations non fonctionnelles telles que la persistance, la sécurité, etc. Le concept de la programmation orientée aspect a été proposé par Kiczales et al en 1997 [KLM+97]. C'est une nouvelle approche de la programmation qui permet de séparer l'implémentation de toutes les exigences, fonctionnelles ou non, d'un logiciel. Le principe est donc de programmer chaque préoccupation *indépendamment* les unes des autres et *indépendamment* de l'application de base et de définir par la suite leurs règles d'intégration en vue de former le système final (figure 1.2.b). Le paradigme de l'AOP propose de structurer les applications sur la base du concept d'aspect et de tissage d'aspects.

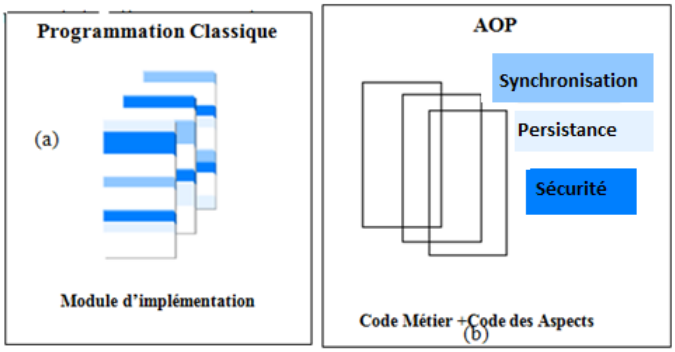


FIGURE 1.2 – (a) Programmation classique (b) La programmation AOP et séparation des préoccupations

Un aspect est une abstraction qui définit une structure et un comportement qui se superposent à l'application (transverse à une application de base). Il est définit par deux partie : *Point de coupe* et *Greffon*. Le *Point de coupe* (*angl Pointcut*) d'un aspect définit d'une manière abstraite l'endroit où le code va être injecté. L'abstraction offerte par les points de coupe va permettre à un aspect d'être tissé à plusieurs endroits d'une même application. Ces points sont appelés des points de jonction et correspondent à des points du flot d'exécution de l'application (invocation de méthode, envoi de message, etc.). Afin de modulariser les préoccupations, les points de coupe sont définis comme des prédicats sur les points de jonction. De ce fait, à un point de coupe correspond zéro, un ou plusieurs points de jonction. La partie greffon (*angl advice*) d'un aspect correspond au traitement ajouté sur un point de coupe qui est le code à injecter. Le greffon spécifie si le code va être injecté avant (*before*) le

point de jonction, après le point de jonction (*after*) ou bien avant et après le point de jonction (*around*) [KHH+01]. L'intégration des aspects à l'application de base est appelée *tissage* et est effectuée par un tisseur.

L'approche aspect s'appuie sur une réalisation par insertion/retrait de code dans l'application. Cependant, d'autres travaux ont une utilisation assez originale du concept aspect en le considérant comme un moyen pour adapter une composition existante d'application (à des endroits bien précis de l'architecture).

1.3.2.2 Spécification de composition : une entité de premier ordre

Initialement le concept d'aspect a été associé à la programmation orientée objet. Par la suite, ce concept a montré un grand intérêt en terme de réutilisation des préoccupations transverses. Donc, il a été couplé à d'autres paradigmes tels que service et composant. Le concept d'aspect a été utilisé dans divers plate-formes pour décrire une composition transverse à un système. Deux possibilités sont offertes : (1) créé une nouvelle composition et/ou (2) s'intégrer dans une composition existante (cela signifie l'adapter). Nous avons choisi de présenter deux approches représentatives de l'utilisation des aspects comme un moyen de spécifier la composition : une approche pour le paradigme service AOP4BPEL [CM04] et une autre pour le paradigme composant AspectOpenCOM [GLTJ08].

AOP appliqué au Service AOP4BPEL [CM04] est un langage orienté aspects de workflow. Ce langage a fourni des mécanismes pour la composition dynamique des web services. Un aspect peut être branché ou débranché dans le processus de composition des services au moment de l'exécution. Par conséquent, il permet de modifier une composition existante de services. Dans ce modèle, un point de jonction est une activité dans un processus BPEL [FBS04]. Le greffon d'un aspect modifie la composition par l'ajout des activités *avant*, *après*, *autour* du point de jonction. AOP4BPEL propose alors l'utilisation d'une entité de premier ordre qui est l'aspect pour décrire une composition de service (ou bien modifier une composition).

AOP appliqué au Composant AspectOpenCOM [GLTJ08] propose une extension du modèle à composant OpenCOM [CBG+08] en lui ajoutant la possibilité de réaliser des compositions par aspects. De ce faite, il est possible de modifier la logique de composition et d'intégrer un nouveau comportement, décrit à l'aide des aspects, dans l'application. La mise en place de ces aspects consiste en l'interception des invocations sur les ports fournis/requis de composants, et en la réalisation de traitements avant ou après.

Dans les deux approches présentées précédemment, les aspects (utilisés pour spécifier les compositions) se basent sur un langage pour décrire les compositions à faire. L'atout de l'utilisation de l'AOP comme un moyen pour spécifier les spécifications de composition est de réduire la réécriture des compositions grâce à l'abstraction offerte par les points de coupe. De cette manière, une spécification de composition peut être appliquée à plusieurs endroits du système et les entités d'adaptation peuvent être spécifiées indépendamment les unes des autres avec le support d'une multi expertise. Un autre avantage est que la spécification de composition peut être transparente pour l'application de base. Nous n'avons pas besoin d'une connaissance complète de l'application de base sur laquelle ces compositions vont être tissées.

Malgré tous ces avantages de l'utilisation du concept d'aspect, des problèmes apparaissent. Les aspects sont spécifiés indépendamment par des concepteurs multiples. Leur intégration dans une même

composition du système logiciel peut faire apparaître des problèmes d'interférence. La gestion des problèmes d'interférence est une étape primordiale dans le processus de la composition pour garantir le comportement du système logiciel. La section suivante présente le problème d'interférence entre les aspects.

1.3.3 Gestion des interférences

Avant de commencer à résoudre le problème d'interférence entre les aspects, il est important de comprendre *quels types d'interférences sont susceptibles de se produire* lors de l'utilisation de l'AOP. Les différents approches regardent le même problème sous des angles différents. Cela est une cause de désaccord entre la plupart des chercheurs abordant ce sujet, chacun définissant sa propre terminologie. Dans ce qui suit, nous dressons une cartographie de la terminologie utilisée ainsi qu'une classification des problèmes d'interférence.

Malgré leurs spécifications indépendantes, les aspects ne sont pas toujours indépendants. En réalité, le développeur a besoin de se préoccuper de ses propres modules (aspects) mais ces derniers sont susceptibles d'interagir avec les aspects d'autres développeurs. A l'issue de ces interactions, l'indépendance des aspects peut être perdue dès qu'ils sont composés. La nature de ces interactions dépend de l'effet de bord des aspects sur le système. Différents termes ont été utilisés pour désigner ces problèmes de composition : *interactions* [STJ+06, GLS+07, MMT06], *interférence* [ARS09, SAM06, MW09, ZCVDBG06], *conflit* [Hav09, DBA07]. Le terme interaction est utilisé dès qu'il y a une relation entre les aspects. Notre expérience montre qu'il existe deux types d'interaction : *positive* et *négative* (nous ne considérons pas les interactions neutres car cela signifie que les aspects sont complètement indépendants). Les interactions positives consolident le bon fonctionnement d'un aspect. C'est la notion de renforcement identifié dans [GLS+07]. À l'inverse, les interactions négatives donnent naissance à un comportement indésirable (ou incorrect) qui perturbe le fonctionnement du système. Dans ce cas, ces interactions sont appelés des *interférences*. Cela signifie que l'exécution d'un aspect influe sur l'exécution des autres. L'interférence est définie dans [DK12] comme suit :

Définition 3 : Interférence
Given a set of aspects, each aspect on its own behaves as expected but when considering all aspects woven together the expected behavior is no longer achieved.

Plusieurs types d'interférences pourront exister dont une sous-partie est désignée par le mot *conflit*. Selon la classification de Tessier [TBB04], il existe quatre types d'interférences qui sont :

Spécifications transverses Les langages utilisés pour la description des aspects et leurs localisations où ils doivent être intégrés peut conduire à deux types de problèmes. Le premier problème se produit lorsque le comportement d'un aspect est intégré au mauvais endroit. Le deuxième problème est la récursivité, c'est à dire que l'aspect intégré à un endroit et le comportement ajouté par l'aspect permettra d'intégrer le même aspect encore une fois et cela d'une manière infinie. Ce type d'interférence est propre à la spécification de l'aspect. Même si l'aspect est le seul à intégrer dans le système, le problème d'interférence pourra apparaître.

Conflits Aspect-Aspect Ce problème se produit lorsque plusieurs aspects co-existent dans un système. Plusieurs types de conflits ont été définis dans cette catégorie : l'exécution conditionnelle (l'application d'un aspect dépend d'un autre aspect qui doit être appliqué pour son bon fonctionnement), exclusion mutuelle (c'est le cas où la composition d'un aspect implique qu'un autre ne doit pas être composé), conflit d'ordre (lorsque les aspect s'appliquent à un même endroit avec le même mot clé (before, after ou around par exemple).

Conflits de type Base-Aspect Un aspect peut également entrer en conflit avec le système de base. Ces types de conflits surviennent lorsque les comportements mis en place dans le système de base ne sont pas cohérents avec les comportements ajoutés par l'aspect. Cela signifie que les problèmes de conflits pourront être présents même si nous avons un seul aspect à tisser dans le système de base.

Conflits de type Préoccupation-Préoccupation Ce type de conflit se produit lorsqu'une préoccupation modifie une fonctionnalité nécessaire pour une autre préoccupation.

Dans la suite de ce manuscrit nous utilisons le terme *interférence* pour désigner les problèmes de composition des aspects. Le fait d'avoir plusieurs types d'interférences implique par conséquent différents approches permettant leur traitement. Nous détaillerons dans l'état de l'art les approches de la littérature pour la détection et la résolution de ces problèmes d'interférences. Nous nous intéressons particulièrement aux interférence de type *Aspect-Aspect* et *Base-Aspect*. En effet, le premier type d'interférence est laissé à la charge du développeur qui doit spécifier rigoureusement les endroits où ses aspects vont être insérés et éviter par la suite le problème de recursivité.

1.3.4 Objectifs de la thèse

La définition d'un mécanisme pour la gestion des interférences permettra d'ajouter des adaptations sans se préoccuper de celles existantes ou de les combiner explicitement. Le mécanisme que nous souhaitons concevoir doit avoir certaines caractéristiques qui lui permettra d'être bien adapté à notre cadre de travail et s'intégrer dans un mécanisme d'auto-adaptation. La figure 1.3 présente ces caractéristiques.

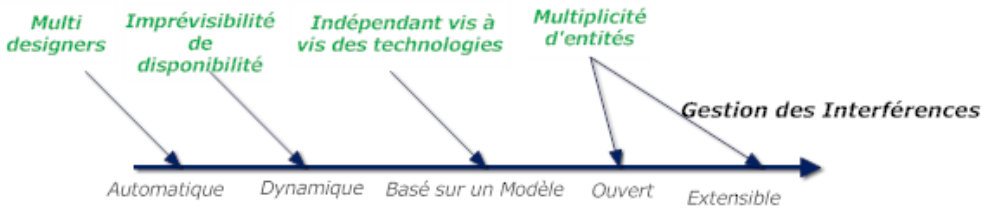


FIGURE 1.3 – Critères pour le mécanisme de gestion des interférences

Nous envisagerons un mécanisme de gestion des interférences qui soit :

Automatique Parce qu'on est dans une approche qui fait intervenir divers designers avec des expertises différentes (ε_4), il n'est pas envisageable de regrouper toutes les entités d'adaptation au sein

d'un seul développeur qui se charge de résoudre tous les problèmes manuellement. De plus, le mécanisme de gestion des interférences doit être intégré dans une boucle d'auto-adaptation. Le mécanisme d'adaptation, et par conséquence le mécanisme de gestion des interférences, doit donc être autonome et doit trouver une solution **automatique** aux problèmes identifiés.

Dynamique Une gestion statique des problèmes d'interférence nécessitera l'arrêt de l'application pour porter la solution d'une manière statique. Cependant en IAm, il n'est pas envisageable de bloquer entièrement une application à chaque fois qu'un problème d'interférence a été identifié. Au contraire, nous souhaitons garder l'application la plus utilisable possible pour une meilleure continuité de service [RTL+10]. Nous nous orientons donc vers une approche de gestion d'interférence **dynamique** qui convient à une adaptation effectuée pendant l'exécution de l'application sans besoin de l'arrêter, et cela en fonction de l'environnement qui est en cours d'évolution (ε_3).

Basé sur des modèles Dans la cas d'une gestion des problèmes basée sur un modèle, le mécanisme s'appuie sur une connaissance du modèle de l'application lui permettant de prendre en considération toutes les informations disponibles pour faire une résolution automatique. L'utilisation de ces modèles offre la possibilité de calculer la solution à un niveau abstrait indépendamment de la technologie d'implémentation et de ne projeter le résultat qu'une fois que la composition d'aspects a été calculée et vérifiée. L'avantage dans ce cas est que l'adaptation ne sera concrète que dans le cas où le résultat est de résolution est pertinent. Réaliser les traitements au niveau d'une représentation abstraite réduit aussi la complexité des calculs de modification [Fer11]. Nous privilégierons alors les mécanismes de résolution **basés sur le modèle de l'application**qui est en cours d'exécution.

Ouvert Le mécanisme de résolution ne doit pas être défini pour un ensemble borné et fixe d'entités d'adaptation. Cela signifie qu'il est toujours possible d'ajouter ou retirer des entités (σ_2) et que le mécanisme doit être capable de calculer une solution.

Extensible Nous envisageons également un mécanisme de gestion des interférence qui soit flexible. C'est à dire qu'il est possible de faire évoluer le mécanisme de gestion des interférences si les besoins des designers changent suite à l'apparition des nouvelles entités logicielles (ε_1) (détails section 3.3.3).

1.4 Plan de la thèse

Ce mémoire est organisé en 3 parties (1.4).

Partie 1 :Introduction et analyse de l'état de l'art

Chapitre 2 ***Analyse de l'état de l'art*** Le chapitre 2 dresse l'état de l'art et analyse les différentes problématiques de la thèse. Nous chercherons tout d'abord à identifier dans la littérature les approches sur lesquelles peuvent reposer les mécanismes de gestion des interférences, qui nous permettront d'apporter des solutions aux challenges que nous avons présentés précédemment.

Partie 2 :Contribution

Chapitre 3 ***La Gestion des Interférences : Un mécanisme général*** A partir des constats de l'analyse de l'état de l'art, nous proposons dans le chapitre 3 un mécanisme de gestion des interférences qui se base sur un formalise de graphe. Nous introduisons dans la première section le modèle de graphe utilisé pour notre raisonnement. Ce modèle sera utilisé par la suite pour exprimer les types d'interférences qui seront adressées dans cette thèse. La dernière section détaillera l'évolution de mécanisme de gestion des interférences que nous proposons qui va intégrer la possibilité d'extension automatique et dynamique du mécanisme.

Chapitre 4 ***Un modèle de composition des applications à deux niveaux*** Dans le chapitre précédent, nous avons introduit les approches de résolution d'une manière abstraite sans présenter le détail du mécanisme permettant une prise en compte des aspects dynamiques et automatiques de cette composition. Ainsi, l'objectif de ce chapitre est de présenter notre approche de composition à deux niveaux. La première section de ce chapitre est une présentation du processus de conception statique du mécanisme de gestion des interférences. Cette étape nécessite l'intervention d'au moins un développeur pour introduire la logique qui sera appliquée par notre mécanisme. La deuxième section détaillera comment ce mécanisme de gestion d'interférences sera appliqué dynamiquement et comment il sera étendu pour répondre aux besoins des développeurs.

Partie 3 : Validation

Chapitre 5 : ***Implémentation et Expérimentation*** Ce chapitre a pour objectif de montrer l'implémentation et l'expérimentation réalisées pour mettre en oeuvre toutes les propositions introduites dans le cadre de cette thèse. Il est structuré en quatre sections principales. Les trois premières sections présentent les choix d'implémentation de notre approche (modèle d'application, les entités d'adaptation et leur processus de tissage). Dans la dernière section, nous décrivons un scénario permettant de dérouler notre mécanisme de résolution des interférences.

Chapitre 6 ***Conclusion et Perspectives*** Enfin, nous concluons ce manuscrit par un rappel des contributions de nos travaux et nous mettons également en évidence les limites et les perspectives d'ouverture de notre étude.

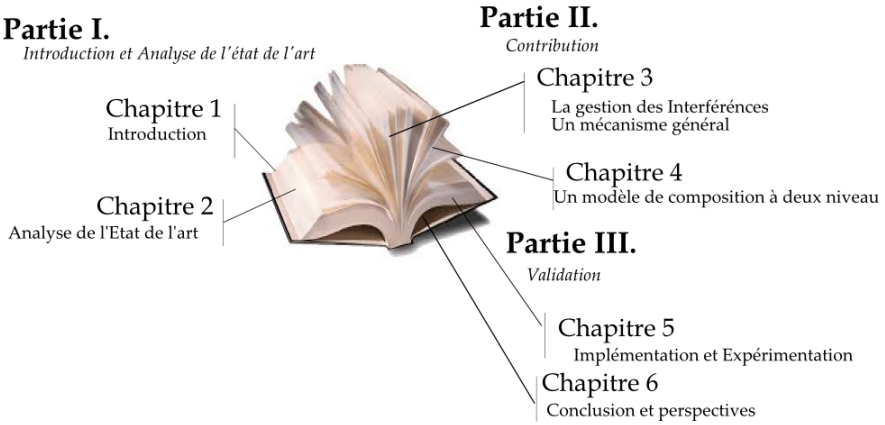


FIGURE 1.4 – Plan du mémoire

CHAPITRE 2

Analyse de l'État de l'art

Sommaire

Les différentes contraintes présentées dans le chapitre précèdent nous ont permis de déterminer les caractéristiques dont doit disposer un mécanisme d'adaptation pour qu'il soit adapté à notre cadre de travail. Nous avons vu que nous avons besoin de disposer d'un mécanisme permettant de réaliser des adaptations compositionnelles (σ_3). Ce type d'adaptation offre au système la capacité d'exploiter dynamiquement de nouvelles fonctionnalités logicielles offertes par des dispositifs qui apparaissent dans l'infrastructure d'une application. La spécification des entités d'adaptation devra répondre aux contraintes imposées par notre cadre de travail. Nous avons montré dans le chapitre 1 que la programmation orienté aspect (AOP) peut être utilisée pour cette spécification. En effet, elle permet d'avoir la séparation entre la préoccupation de la composition (adaptation) et la logique applicative avec le critère d'*indépendance* de spécification (ε_4).

Lorsque des adaptations séparées dans des aspects sont appliquées, elles doivent être composées. À l'issu de cette composition, des problèmes d'interactions peuvent apparaître entre eux. Lorsqu'au moins un des ces aspects ne fonctionne plus comme prévu, on parle d'**interférence** [ARS09]. Notre mécanisme de composition doit considérer ces problèmes et proposer une résolution adaptée. Nous étudierons alors les différentes approches pour la détection et la résolution des interférences. Nous présentons deux grandes catégories d'interférences : sémantiques et syntaxiques. Nous verrons par la suite que deux types d'approches de résolution ont été proposées dans la littérature : des approches de résolution externes qui visent à trouver un ordre entre les aspects et d'autres approches qui proposent une résolution interne qui fusionne les comportements des aspects en interférence. Nous montrons que la *résolution interne* correspond au mieux à nos besoins.

La résolution interne des interférences se base sur la composition des opérateurs utilisés dans la spécification des aspects. Ces opérateurs correspondent aux connecteurs d'assemblage des entités logicielles composant l'application. Par conséquent, nous nous intéressons dans la deuxième section de ce chapitre aux modèles de connecteurs ainsi qu'aux approches de leur composition.

L'étude de ces approches existantes nous permettra de souligner leurs limites et d'identifier les caractéristiques que doit respecter un mécanisme de gestion des interférences destiné à être utilisé en IAm. Nous détaillons donc dans la dernière section du chapitre nos objectifs et les critères qui devront être fournis par notre mécanisme de gestion des interférences.

2.1 Les problèmes de la composition : les interférences

Lorsque plusieurs aspects sont tissés à un système de base, les interactions doivent être soigneusement analysées. Nous nous intéressons aux interactions qui ont des effets négatives sur le système. C'est le problème d'*interférence*. De nombreux travaux sont consacrés à la détection et/ou la résolution des interférences entre les aspects. Dans la suite, nous considérons la classification de Chengwan [HLH08] qui divise le problème en deux catégories : *syntaxiques* et *sémantiques*.

2.1.1 Approches syntaxiques

Une interférence syntaxique apparaît lorsque deux ou plusieurs aspects partagent un point de jonction ou bien *des ressources* de l'application de base. La superposition des aspects à un système de base peut être comportementale et/ou structurelle. La superposition comportementale se réfère à l'ajout du comportement exprimé dans les parties greffons. La superposition structurelle prend plusieurs formes de transformations du système de base, généralement par l'ajout d'éléments dans les systèmes tels que des méthodes et des variables. Un aspect peut être constitué d'une combinaison de superpositions comportementales et structurelles. Dans cette section, nous étudierons quelques travaux pertinents pour la détection et/ou la résolution des interférences syntaxiques. Nous avons classé ces travaux selon leurs stratégies de résolution des interférences qui peuvent être : *externe* ou *interne*. La figure 2.1 illustre la différence entre ces deux stratégies. Dans une résolution externe, les comportements introduits par les aspects restent intacts et la solution consiste à trouver le bon ordre d'exécution. À l'inverse, une résolution interne mélange les comportements des aspects dans le but de trouver le comportement souhaité. Dans ce qui suit, nous présentons en détails des approches faisant partie de chacune de ces deux catégories.

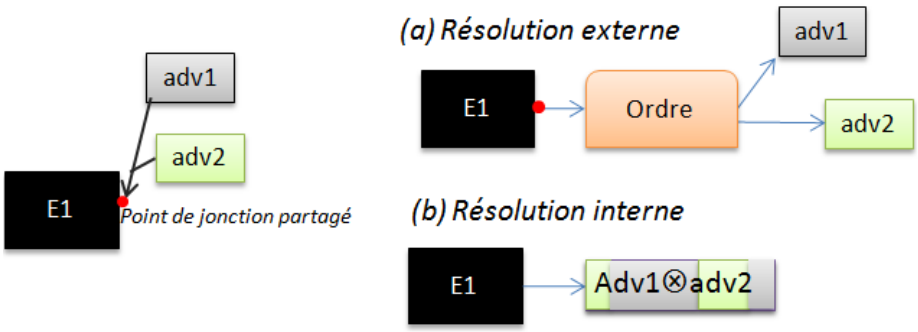


FIGURE 2.1 – (a) Résolution externe et (b) la Résolution interne

2.1.1.1 Résolution externe

Nous désignons par le terme résolution d'interférence externe, la spécification d'un ordre entre les greffons qui s'appliquent à un même endroit sans modifier le comportement de chacun d'entre eux. Cet ordre est une séquence d'exécution unique pour éviter le problème d'interférence. Plusieurs stratégies peuvent être utilisées pour déterminer cet ordre tels que : précédence, priorité statique (AspectJ [KHH+01]), priorité dynamique (Prose [PGA01]), sémantique d'opérateurs (opérateur entre les aspects EAOP [DS02]), etc. Dans la suite nous détaillons quelques approches qui ont utilisé cette méthodologie de résolution des interférences. Nous les classons selon le moment où la résolution des interférences est effectuée, c'est à dire avant l'exécution de l'application (*tissage statique*) ou bien pendant l'exécution (*tissage dynamique*).

Approches de résolution statiques

Un tisseur est dit statique s'il intègre tous les aspects dans l'application avant son exécution. De ce fait, les interférences sont détectées et résolues avant de passer à l'exécution. L'exemple classique de ce type de tissage est le tisseur dans AspectJ [KHH+01]. Ce tisseur intervient à la compilation et intègre le code des greffons dans le code de l'application de base. Les problèmes d'interférences identifiés à la compilation sont résolus par la définition *explicite* d'un ordre entre les aspects concernés.

D'autres approches ont proposé un traitement précoce des problèmes d'interférences. Les travaux qui seront présentés dans la suite, se placent plutôt juste après la phase d'étude des exigences (nous ne présentons pas les travaux qui s'intéressent aux problèmes d'interférence au niveau de l'étude des exigences). La gestion des interférences peut être alors anticipée et effectuée pendant la phase de conception du système avant même de passer à son implémentation. La différence majeur dans ce cas est que nous n'avons pas l'accès au comportement complet de l'application qui est fourni par le code source. Par conséquence, les techniques proposées pour la détection des interférences au niveau du code de l'application (programming level) ne peuvent pas être utilisées telles quelles pour ce genre d'approche. Pour ce faire, plusieurs travaux reposent sur la technique d'AOM (Aspect Oriented Modeling) [SSK+07] pour tisser les aspects au niveau du modèle de l'application. Nous allons présenter trois de ces approches en insistant sur les mécanismes de gestion des interférences qu'elles proposent. Une synthèse sur l'étude de ces travaux sera également proposée.

Zhang et al [ZCVDBG06] proposent de tisser des aspects dans les modèles UML exécutables ce qui permet la génération automatique du code de l'application finale. Les parties point de coupe et greffons d'un aspect sont spécifiées à l'aide de machines à états finis. L'interférence traitée est de type *flot de contrôle* (définit section 2.2.2) tel que par exemple, deux actions *skip* et *proceed* au niveau d'un point de jonction partagé. L'objectif visé est de réduire au maximum l'occurrence des problèmes d'interférences en déterminant automatiquement l'ordre dans lequel les aspects doivent être tissés. Cet ordre est calculé en se basant sur les directives spécifiées par les développeurs pour chacun des aspects. Plusieurs directives ont été définies tel que par exemple la *dépendance* (un aspect A3 ne sera appliqué que lorsque les deux aspects A2 et A1 seront appliqués à ce même point de jonction) et le *masquage* (un aspect A2 ne sera pas appliqué si l'aspect A1 a été appliqué au niveau d'un point de jonction partagé entre les deux). La limite principale de cette approche est qu'il n'est pas toujours garanti de trouver une solution si les directives n'ont pas été déjà définies explicitement par les développeurs. Dans un tel cas, un message d'erreur est remonté aux développeurs et le tissage échoue.

Une autre technique largement utilisée pour la détection des problèmes d'interférences est la théorie de transformation des graphes [ENRR88] (plus de détail dans la section 3.3.1.1). *Mehner et al* [MMT06] ainsi que *Whitle et al* [WJ08] définissent un aspect comme étant une règle de transformation de graphe qui va être appliquée au diagramme d'activité UML ou diagramme d'objet. Une règle de transformation de graphe se compose de deux parties : une partie gauche et une partie droite. Si le sous graphe de la partie de gauche de la règle a été identifié dans le diagramme d'activité UML, alors il sera remplacé par le sous graphe de la partie droite de la règles. Ce formalisme permettra d'avoir des points de jonction sous la forme d'un groupement d'objets avec leur liens (ou groupement d'activités). Comme un aspect, une règle de transformation pourra ajouter des objets, modifier les valeurs des attributs, etc. Dans [MMT06], outre les parties points de coupe et greffon, un aspect (sous forme de règle) contient une partie *condition* qui est utilisée pour déterminer quand il pourra être tissé. Puisque les aspects sont représentés par des règles de transformation de graphe, le problème d'interférences est traité en utilisant les techniques définies dans ce domaine. En effet, il existe déjà des outils dédiés à l'analyse des interactions entres les règles de transformation de graphe. La technique utilisée est *Critical Pair Analysis* (CPA). Cette technique permettra de détecter deux types d'interférence : *les dépendances* (une règle est activée après l'application d'une autre) ainsi que les *conflits* (l'application d'une règle désactive une autre). La figure 2.2 montre le résultat d'un exemple d'analyse des

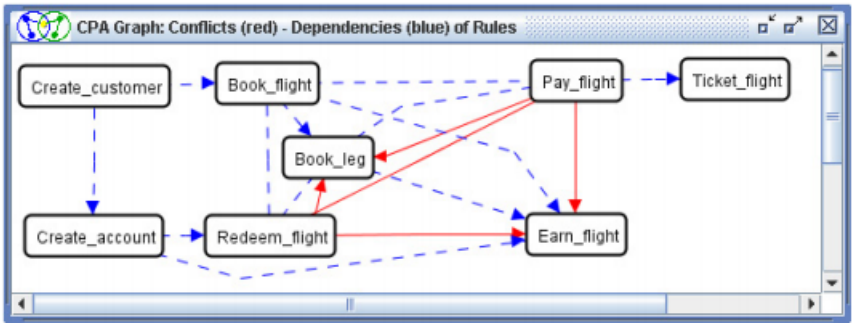


FIGURE 2.2 – Analyse des interactions entre les règles de transformation à l'aide de CPA

interactions entre des règles de transformation à l'aide de la technique CPA. Les rectangles arrondis représentent les règles de transformation et les flèches leurs interactions (conflit en couleur rouge et dépendance en bleu). Le traitement des problèmes identifiés dépend de l'application : ils peuvent être tolérés ou bien doivent être résolus (pour obtenir un comportement cohérent). En effet, certains des conflits identifiées conduiront à des modifications des règles de transformation. La solution est fournie par le développeur qui détermine l'ordre de tissage des aspects qui va être utilisé en cas de problème.

Dans [CHA+10] une représentation à l'aide du modèle de graphe a été utilisée. La différence par rapport aux approches présentées précédemment est qu'un graphe représente l'espace d'états d'un programme en faisant varier l'ordre dans lesquels les aspects sont tissés. Une interférence est détectée si l'ordre d'application des aspects met le système dans un état final différent. La figure 2.3 montre un exemple de tissage de deux aspects et l'espace d'état généré au niveau d'un point de jonction partagé. La racine du graphe présente une expression qui sera évaluée. Deux sous branches seront parcourues : True et False. Sur le côté droit (où la condition est False), suite au tissage des aspects selon un ordre différent l'état final du système est le même ($s54$). Sur le côté gauche (où la condition est vraie), les branches issues de la racine ne se fusionnent pas parce qu'un aspect a empêché l'autre de s'appliquer. Il s'agit alors d'un problème d'interférence.

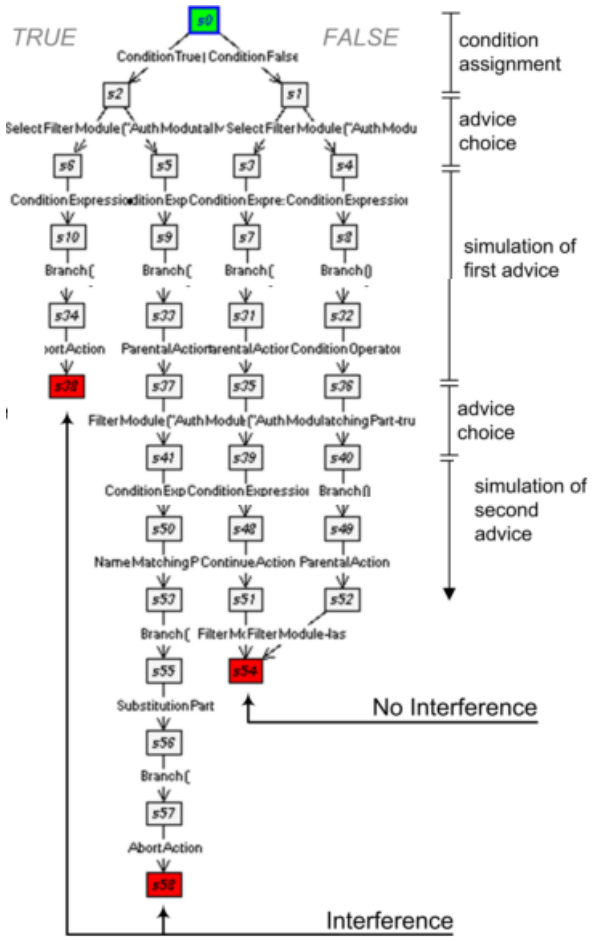


FIGURE 2.3 – Tissage d'aspect selon différents ordre et espace d'état de l'application

L'avantage de cette approche est l'utilisation d'un modèle abstrait de l'application pour la détection des interférences ce qui permettra d'avoir une approche indépendante d'un langage d'aspect en particulier. Cependant, les problèmes d'interférences ne se limitent pas au tissage des aspects dans un mauvais ordre. Même si la variation d'ordre de tissage des aspects donne le même résultat, ceci ne garantit pas la cohérence du résultat final. Une autre limite est que la technique proposée permettra seulement la détection des problèmes d'interférences mais il n'y a pas de proposition de méthodologie de résolution de ces problèmes.

↪**À retenir.** L'utilisation d'un modèle abstrait d'application permet de décrire la problématique d'une façon générique. De cette manière, la résolution proposée pourra être appliquée à des applications basées sur des technologies et langages différents. Les approches présentées précédemment ont proposé d'utiliser une modélisation sous forme de graphes. Cette représentation a permis de décrire à la fois les applications et d'utiliser la théorie des graphes pour l'identification des interférences. La technique de transformation de graphe a été utilisée pour simuler le tissage d'aspects ainsi que l'analyse de leurs interactions possibles (dépendance et conflit).

Toutes les approches de résolution présentées dans cette section reposent sur la définition explicite et figée par le développeur d'un ordre entre les aspects en interférence. Cependant, il est difficile de gérer l'ensemble de la combinatoire des relations de dépendance lorsque le nombre d'aspects à tisser devient élevé. Cela nécessite aussi la connaissance de tous les aspects qui vont être intégrés dans le système avant même son exécution. Toutefois, nous avons vu dans le chapitre 1 que de nouvelles fonctionnalités et besoins peuvent apparaître pendant l'exécution et seront par conséquent ajoutées sans arrêter le système (σ_2). Ce type d'approche limite alors la capacité du mécanisme de composition à prendre en considération cette imprévisibilité (ε_3). Les techniques utilisées pour la détection ainsi que la résolution des interférences devront faire appel à des moyens plus flexibles qui considèrent le caractère dynamique dans lequel le système s'exécute. Nous nous orientons vers la gestion des interférences la plus dynamique possible car il n'est pas envisageable de bloquer entièrement une application à chaque apparition d'un problème.

Maintenant que nous avons identifié qu'il est intéressant d'utiliser une modélisation abstraite de l'application pour raisonner sur les problèmes d'interférences et que la résolution devra être effectuée pendant l'exécution, nous allons étudier les approches de traitement dynamique d'interférences émergeant dans la littérature.

Approches dynamiques

Douence et al. [DS02] adressent le problème d'interférence entre aspects aux points de jonction partagés pendant l'exécution. Ils ont défini un langage formel EAOP (Event-based Aspect-Oriented Programming) qui intègre un support pour la résolution des conflits. L'approche proposée est basée sur des faits observables qui sont les événements d'exécution. Les aspects peuvent être en effet vus comme des propriétés sur les traces d'exécution du programme et sont exprimées au moyen de ces événements. Des opérateurs de composition explicites ont été proposés pour résoudre les interférences qui sont : *séquence*, sélection d'un unique aspect, séquence arbitraire décidée par l'analyseur. Ce travail a été ensuite étendu pour supporter *les aspects concurrents* [DFL$^+$06]. Malgré le support de la résolution pendant l'exécution, la définition de la solution est figée et ne pourra pas être modifiée pendant l'exécution.

Durr et al [DBA07] proposent la représentation de tissage d'aspects à l'aide d'un modèle de graphe. Ils modélisent chaque point de jonction comme étant une ressource manipulée par les différents greffons qui partagent ce point de jonction. Un ensemble d'opérations sur les points de jonctions sont définies comme par exemple lire (*write*) et écrire (*read*). L'approche proposée traite deux types de conflit : conflit de donnée et conflit de flux de contrôle. Un conflit de données est lié à la modification d'une des propriétés de ressources (par exemple une séquence *write-read* du même paramètre du point de jonction). Le conflit de flux de contrôle est lié aux actions introduites par les greffons sur les ressources (par exemple la séquence *skip-proceed*). Pour la détection des conflits, les actions des greffons sont appliquées au niveau de chaque point de jonction.

Cela va générer un graphe représentant la trace d'exécution de l'application (une machine virtuelle est utilisé pour suivre à l'exécution les différentes actions faites par les greffons sur les ressources à chaque point de jonction). Chaque noeud du graphe obtenu représente un élément de programme à

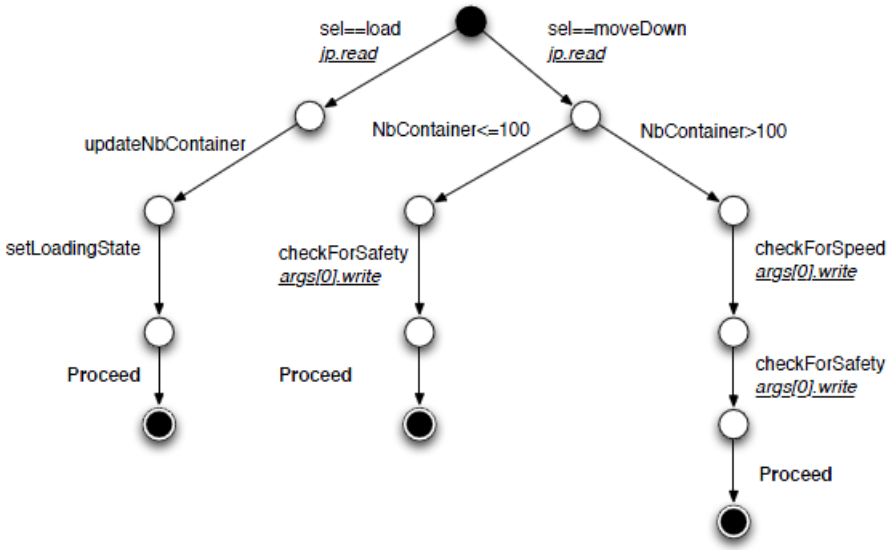


FIGURE 2.4 – Un extrait du graphe de trace des actions entre deux aspect (SaveEnergy et TruckSafety)

évaluer (c'est à dire, point de jonction, les variables de données, etc.). Chaque chemin dans le graphe illustre une séquence possible des actions réalisées par des greffons à un point de jonction commun. Les arcs dans le graphe sont étiquetés avec des opérations sur les ressources (read ou write) et des actions des greffons (proceed, skip, etc.). La figure 2.4 illustre un exemple de graphe obtenu pour un point de jonction. Les règles utilisées pour l'identification des conflits ont été prédéfinies par le développeur. Une règle définit une séquence d'actions qui produit forcement un problème de conflit comme par exemple deux actions *write* sur une même variable (illustré dans la figure précédente). Quand une règle de conflit est identifiée dans le graphe de la trace des actions, le problème est signalé à l'utilisateur. Cette approche présente un inconvénient majeur. Les règles de détection des conflits doivent être explicitement déterminées par l'utilisateur ce qui reste une opération complexe et coûteuse en temps.

Cette limitation a été résolue dans [MW09]. Les règles de définition des conflits ont été remplacées par des intentions de composition (*compositional intentions*). Ces intentions de composition sont définies pour chaque aspect et décrivent des hypothèses sur les actions d'autres aspects sur des points de jointure partagés. Chaque intention de composition a un type et une description du comportement. Le type d'intention de composition décrit comment un greffon doit être exécuté par rapport aux autres greffons. En plus des directives traditionnelles *before* et *after*, la directive *ignored* a été définie pour indiquer que le greffon doit être ignoré lorsque la partie de description du comportement (dans l'intention de composition) a été spécifiée par d'autres aspects. Une description du comportement est une conjonction et/ou d'une disjonction de prédicats d'action. La plupart des actions définies sont de type flot de contrôle (proceed, skip, etc.). Cependant, il y a aussi deux actions de flot de donnée qui sont *read* et *write*. *La résolution des interférences selon cette approche est spécifiée dans l'aspect lui-même. L'intégration d'un nouvel aspect dans le système nécessite alors la révision de tous les aspect existants. Nous perdons alors l'atout de la séparation des préoccupations avec l'indépendance de spécification qui a été une des motivations de l'utilisation du concept d'aspect.*

D'autres approches proposent la définition d'une stratégie de résolution plus évoluée que la simple précédence entre aspects. Dans [GLS+07], la solution proposée consiste en l'utilisation des contrats (interaction contracts) qui sont consultés par le tisseur à l'exécution pour respecter les relations ainsi définies entre les aspects. Divers types de contrat ont été définis dont chacun est relatif à un type particulier d'interférence. Les types de problèmes supportés dans cette approches sont : *conflit* (un aspect à besoin de l'absence d'un autre pour fonctionner correctement), *dependence* (un aspect a besoin d'un autre pour bien fonctionner), *précédence* (les aspects doivent respecter un ordre) et *résolution* (un ensemble d'aspects incompatibles peut exiger la présence d'un aspect de la résolution afin de leur permettre de coexister dans un même système). Un exemple de contrat de type résolution est défini comme suit :

instance['transaction']=resolve(instance['distribution'] AND instance ['persistence']

Cela signifie que l'aspect de transaction n'est nécessaire que lorsque les aspects de distribution et la persistance sont appliqués aux mêmes endroits. En spécifiant explicitement ces relations dans des contrats externes, le couplage entre les aspects est réduit (contrairement à [MW09]). Un aspect particulier est ensuite utilisé pendant l'exécution pour vérifier les violations de contrats. *Même si les contrats sont utilisés à l'exécution, la solution donnée reste explicite puisqu'elle nécessite l'intervention du concepteur de l'application. De plus, nous pouvons avoir un cas où les contrats spécifient des stratégies contradictoires. Dans ce cas de figure, ils sont tous ignorés et on perd alors les comportements spécifiés par ces aspects.*

Les approches présentées ci-dessus présentent deux problèmes majeurs. Tout d'abord, les stratégies de précédence entre les aspects ne peuvent pas être modifiés dynamiquement au cours de l'exécution de l'application. Deuxièmement, il y a une rupture technologique entre la façon de spécifier des aspects et la façon dont les conflits sont résolus. La contribution originale de *Dinkelaker et al* [DMB09] permet de préciser la logique de résolution dans un même langage que celui de spécification des aspects. Le langage proposé est *Meta Aspect Protocol* [DMB09] et permet à l'utilisateur de définir une stratégie de résolution des interactions dites *context dependent* [STJ+06].

Illustrons ces propos par un exemple qui utilise cette approche de résolution. Un premier aspect *Alice_to_AMAlice* permet de rediriger les appels téléphone de *Alice* vers sa boîte vocale si elle n'est pas disponible. Un autre aspect *Alice_to_Bob* de priorité plus élevée que le premier permet de rediriger cette fois les appels téléphoniques d'*Alice* vers le numéro de *Bob* si elle ne répond pas. Un troisième aspect *Bob_to_AMBob* redirige les appels téléphoniques de *Bob* vers sa boîte vocale lorsqu'il est indisponible. Le tissage de ces trois aspects (illustré par la figure 2.5) va enregistrer les appels téléphonique de *Alice* sur la boîte vocale de Bob si tous les deux sont indisponibles. Cependant Alice souhaite que les message soient enregistrés sur sa boite vocale dans un tel cas.

Ce type d'interaction ne peut pas être résolu par un ordre prédéfini et nécessite plutôt la modification dynamique de la stratégie de la composition des aspects en fonction du contexte de l'application en cours d'exécution. Pour cet exemple, le bon ordre des aspects dépend d'une condition dynamique qui est si le répondeur de *Bob* est actif ou non. Pour ce faire, des comparateurs consultables et modifiables à l'exécution ont été proposés. Ces comparateurs (qui sont des modèles de stratégies ordonnant les aspects) peuvent être complexes c'est-à-dire combiner plusieurs stratégies selon les variables liées à l'environnement.

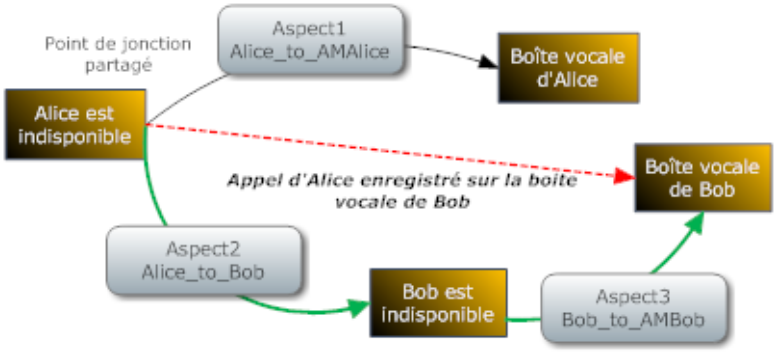


FIGURE 2.5 – Tissage des aspects : *Alice_to_AMAlice*, *Alice_to_Bob* et *Bob_to_AMBob*

Toutes les approches que nous avons étudiées jusqu'à présent permettent de détecter principalement deux types d'interférence : Dépendance *: un aspect a besoin d'un autre pour bien fonctionner (ou bien l'absence) et* Conflit *: un aspect en active ou désactive un autre. Plusieurs travaux ont été basés sur une représentation de l'application à l'aide des graphes ce qui rend la solution proposée indépendante d'une implémentation particulière. La résolution proposée (dans le cas où elle existe) consiste à restreindre le flot de contrôle pour résoudre le problème. Cette restriction consiste à déterminer un ordre entre les aspects soit d'une manière ad-hoc ou bien automatique. La spécification de la précédence entre les aspects est le moyen intuitif pour composer des aspects, malheureusement, cela ne résout pas toutes sortes d'interférences. Le fait d'imposer un ordre implique de trouver cet ordre total. Certains problèmes d'interférences peuvent survenir quelque soit l'ordre de composition choisi. À titre d'exemple, nous supposons que nous avons deux aspects : un aspect temps réel qui impose certaines contraintes de temps sur certaines actions et deuxième aspect qui impose une contraintes de synchronisation sur ces même actions. Le problème est que nous ne pouvons jamais garantir que le délai est respecté à cause du blocage potentiel.*

↪**À retenir.** Nous avons besoin d'une approche de détection et de résolution des interférences *non anticipée*, *dynamique* et reposant sur un *modèle abstrait* de l'application. L'utilisation du formalisme de graphe a montré un grand intérêt et une flexibilité dans la solution proposée. Nous avons vu aussi que la résolution s'appuyant sur la définition d'un ordre entre les aspects ne résout pas tous les cas d'interférences.

Dans la suite, nous nous intéressons à d'autres approches qui font la résolution des interférences d'une manière interne aux aspects en fusionnant leurs comportements. Ce type de résolution est qualifié d'interne car les aspects sont combinés pour en former un nouveau. Dans cette catégorie d'approches, l'état de l'application à un instant donné ne doit pas dépendre de l'ordre d'application des aspects. Dans un même cycle de tissage, quelque soit l'ordre d'application des aspects le résultat sera le même.

2.1.1.2 Résolution interne par fusion des comportements

La fusion comportementale des greffons en conflit est rendu possible grâce à la propriété *grey box* de ces derniers qui signifie que la sémantique d'une partie du greffon est connue. Étant donnée cette connaissance, plusieurs approches proposent de travailler sur une manière de les composer avec une résolution d'interférence semi-automatique [Mos10] ou automatique [CFW09].

ADORE (an Activity meta moDel supOrting oRchestration Evolution) [Mos10] est une plate-forme qui permet la composition automatique des services en utilisant une approche dirigée par les modèles. Il prend en charge la définition statique de modèles de comportement, et leurs compositions. Leur motivation est d'être indépendant des langages qui peuvent exprimer les processus métiers pour la composition des applications. Chaque modèle décrit un morceau d'orchestration de services (appelés fragments d'orchestration) qui seront composés pour produire un modèle décrivant une orchestration d'un ensemble plus large. Les fragments décrivent différents aspects d'un processus métier complexe, où chaque aspect répond à une préoccupation et sont insérés avant (before) ou après (after) une activité. Dans le cas où il y a un ordre à respecter entre les aspects, ils sont appliqués dans des cycles successifs. Dans un même cycle le résultat de la composition des aspects est indépendant de tout ordre. Ces aspects sont fusionner pour former un nouvel aspect. ADORE intègre un algorithme pour la détection et la gestion des interférences lors de la composition des fragments sur un cycle. Une interférence est détectée au niveau du modèle lorsqu'il y a un accès concurrent à une variable ou bien des exceptions concurrentes. Dans un tel cas, le développeur intervient pour rectifier le résultat de la composition automatique afin d'avoir le comportement souhaité. Malgré la réutilisation de la solution proposée par le développeur pour la résolution d'autres cas similaires, la solution proposée reste semi-automatique. Nous avons besoin de définir *une approche de résolution automatique* sans avoir recours au développeur pour résoudre les interférences apparues lors de la composition du système.

Pour résoudre les interférences durant l'exécution du système, plusieurs approches proposent de travailler au niveau d'une représentation de son architecture d'exécution. La représentation du système pourra être faite en se basant sur son modèle de flot de données ou bien son modèle de flot de contrôle. Le modèle du flot de contrôle a été utilisé par Daniel Cheung-Foo-Wo[CFW09] pour la détection et la résolution des interférences sans interrompre l'exécution de l'application. Il utilise une représentation sous forme d'arbre syntaxique de la partie greffon des aspects (un flot de contrôle). La solution proposée repose sur la connaissance de la sémantique du greffon qui est considérée en tant que boîte blanche. Cela permet au tisseur de résoudre automatiquement les problèmes en utilisant un raisonnement logique qui fusionne les greffons aux endroits où il y a interférence. Les règles de fusion ont été construites à partir du modèle de fusion d'interaction du Berger [Ber01]. Dans un premier temps, les auteurs ont proposé de gérer *la concurrence entre les aspects*. Deux aspects sont concurrents lorsqu'ils s'appliquent dans un même point de jonction qui est dans ce cas un composant. Pour résoudre ce problème, les arbres de flot de contrôle de chacun de ces aspects vont être fusionnés pour n'en garder qu'un seul (un seul flot de contrôle à partir d'un port de composant). Le deuxième type d'interférence traité par les auteurs est l'*accès concurrent à un composant* mais cette fois ci la solution proposée est locale. Pour ce faire, ils proposent d'autres opérateurs de coordination avec un ensemble de règles logiques pour leur fusion. Dans l'approche proposée, le tisseur est capable de traiter soit le premier type d'interférence ou bien le deuxième mais pas les deux à la fois.

Le mécanisme de fusion proposé est basé sur les opérateurs du langage ISL4WComp[CFW09]. Les règles d'interactions ont été formalisées par des arbres syntaxiques représentant un flot de contrôle. Fusionner deux arbres consiste **à fusionner les opérateurs** en utilisant un ensemble de

règles logiques prédéfinies. L'avantage d'utiliser un tel mécanisme de fusion est de rendre la résolution des interférences automatique sans avoir besoin de l'intervention du développeur. Ceci est important dans le cadre de notre travail pour s'intégrer dans des mécanismes assurant l'auto-adaptation. Deux limitations sont présentes dans cette approche : (1) malgré la capacité de résoudre les deux types de problème, le tisseur ne pourra résoudre que l'une de deux à la fois et (2) la solution proposée utilise un ensemble prédéfini d'opérateurs liés au langage utilisé pour la spécification des compositions. L'ajout d'un opérateur n'est possible que statiquement. C'est une tâche manuelle et assez complexe car elle nécessite de déterminer comment composer ce nouvel opérateur avec tous les autres opérateurs déjà existants. L'introduction d'un nouvel opérateur à l'exécution rend les problèmes d'interférences non resolvable. La deuxième limitation est un vrai obstacle dans notre cadre de travail puisque la multitude d'entités logicielles et leur variabilité pourront avoir comme conséquence le besoin d'introduire des nouveaux opérateurs.

Toutes les approches présentées jusqu'à présent s'intéressent aux problèmes d'interférence au niveau des points de jonction partagés. Cependant, une interférence peut avoir lieu entre les aspects sans avoir forcément un point commun. Ce sont les interférences sémantiques qui feront l'objet de la section suivante.

2.1.2 Approches sémantiques

L'interférence sémantique apparaît lorsque les comportements introduits par les aspects sont contradictoires soit par rapport à l'application de base ou bien par rapport aux autres aspects. Pour pouvoir traité un tel type d'interférence, le modèle d'aspect doit être enrichi par une représentation formelle de son comportement. Katz et al [KK08] proposent l'utilisation de LTL (Linear Temporal Logic) pour décrire les propriétés associées à un aspect sous forme *«assume-guarantee»*. Si la partie *«assume»* n'est pas satisfaite par le programme de base, l'aspect ne peut pas être tissé. La partie *«guarantee»* doit être satisfaite après le tissage. Cependant, certains aspects sont conçus pour changer le comportement du système de base. Dans ce cas, le concepteur doit spécifier explicitement quelles sont les propriétés qui ne doivent pas être modifiées pour son système de base. Par conséquent, les propriétés du système de base doivent être bien définies et le concepteur doit avoir une connaissance précise du système à construire.

Hannousse et al [HDA11] ont enrichi le langage ADL avec des informations nécessaires pour détecter les interférences entre les aspects tissés à un système à composants. Ils ont unifié et explicité le comportement des composants primitifs et des aspects sous forme de machine d'états finis. L'ADL a été enrichi par un ensemble de règles de tissage pour préciser comment tisser et composer les aspects au système de base. Toutes ces représentations nécessitent une certaine connaissance et maîtrise des applications de base sur lesquels les aspects vont être tissés. Il est indispensable de spécifier explicitement le comportement attendu du système après l'opération de tissage ainsi que le résultat attendu par chaque aspect. Une interférence est détectée si l'une des propriétés des aspects tissés et celles du système de base qui doit être préservée après le tissage des deux aspects, n'est pas vérifiée. Dans ce cas, la trace d'exécution est remontée au développeur qui doit déterminer la source d'erreur et de trouver la stratégie de composition résolvant le problème. Un premier pas vers l'automatisation de la résolution est la définition d'un catalogue d'**opérateurs de composition** sous forme de patrons décrivant la structure abstraite de chaque opérateur (machine à états finis abstraite). La résolution d'interférence proposée est une extension de celle qui a été proposée par [DS02]. La sémantique de chaque opérateur

est définie à l'aide d'un tableau spécifiant la poursuite ou l'ignorance de chaque point de jointure selon son type. Par exemple, la composition séquentielle entre *skip* et *proceed* : $SEQ(skip, proceed)$ est *proceed*. Nos entités logicielles sont des boîtes noires donc on ne connaît pas a priori la sémantique des entités logicielles qui forment nos applications. De ce fait, nous ne maîtrisons pas le comportement du système final après une adaptation (aussi à cause de l'imprévisibilité et la multitude d'entités impliquées). Ces faits constituent un frein pour l'application de cette approche dans notre cadre de travail.

De Fraine et al [DFQJ08] proposent une analyse statique en se basant sur une représentation du flot de contrôle pour détecter les interférences. Cette représentation leur a permis de détecter les interférences sans se limiter seulement au problème au niveau des points de jonction partagés. L'approche proposée se base sur la définition d'un ensemble de prédicats qui expriment les relations en terme de flot de contrôle entre les aspects et le programme de base. Les graphes de flots de contrôle sont générés automatiquement suite à une analyse statique du code du programme après le tissage de chacun des aspects. Ces graphes sont parcourus pour déterminer pour chaque méthode les différents chemins possibles à partir de son entrée jusqu'à sa sortie. Ces chemins sont utilisés pour évaluer les différents prédicats. D'autres approches ont montré qu'il est insuffisant de travailler sur une représentation en terme de flot de contrôle et qu'un modèle doit aussi bien intégrer le flot de données dans l'analyse des interférences [WTR07].

Weston et al [WTR07] proposent AIDA qui est une extension du compilateur AspectJ en lui intégrant une analyse basée sur le flot de données. Une interférence est détectée si une variable utilisée par un aspect et modifiée par un autre. Pour ce faire, AIDA effectue une analyse avant et après le tissage. Au stade pré-tissage, le graphe de flot de contrôle est analysé en utilisant deux fonctions appelées *transfert et summary*. La fonction *transfert* indique les effets de chaque instruction sur les variables de données alors que la fonction *summary* indique l'effet de chaque appel de méthode. Après le tissage des aspects, ces deux fonctions sont recalculées et comparées au résultat obtenu pendant la phase de pré-tissage. Ce résultat est utilisé pour détecter les interférences potentielles relatives aux données. Aucune méthodologie de résolution des problèmes n'a été proposée.

2.1.3 Synthèse : Gestion des interférences

Dans notre travail, nous nous intéressons à la détection et la résolution des interférences qui se produisent lors du processus de composition. Le tableau 2.1 illustre un résumé des techniques utilisés par les travaux étudiés. Dans la suite, nous listons un ensemble de conclusions à partir des approches étudiées.

2.1.3.1 La détection des interférences

L'approche de détection d'interférence que nous proposons devra respecter l'ensemble de contraintes déjà présentées en section 1.1.2 et devra assurer les même caractéristiques que celle du mécanisme d'adaptation (section 1.2.2) dans lequel elle va s'intégrer. La détection des interférences doit intervenir pendant l'exécution du système et doit accepter l'ajout ou bien le retrait des aspects (σ_2). Nous avons vu que l'utilisation d'une représentation abstraite de l'application offre divers avantages (être indépendant des langages et des technologies d'implémentation, fournir un cadre formel, etc.). Notre approche de détection utilisera donc un modèle abstrait d'application pour identifier les problèmes. Dans tous les approches présentées précédemment, la détection des interférences se base

sur une analyse d'une représentation du système qui peut être au niveau du flot de contrôle ou du flot de données. Dans notre cadre de travail, les entités logicielles utilisées pour composer le système sont des boîtes noires (ε_2). Nous n'avons pas accès à leur modèle de flot de données. Par conséquent, nous utilisons dans notre raisonnement une représentation de flot de contrôle du système. Comme dans [MMT06] et [CHA+10], nous utilisons le formalise de graphe qui offre un cadre formel riche. Nous mentionnons que notre approche ne repose ni sur un modèle à composant spécifique, ni sur un modèle de graphe particulier.

Nous avons vu qu'il existe principalement deux catégories d'approches pour gérer le problème d'interférence : syntaxique et sémantique. Pour pouvoir détecter les interférence d'ordre sémantique, il est indispensable d'enrichir le modèle de l'application en y ajoutant le comportement existant ainsi que le comportement souhaité suite au tissage des aspects. Dans notre cadre d'étude, la notion d'application de base n'existe pas car l'application se construit d'une manière opportuniste en fonction des entités logicielles disponibles à cet instant (ε_3). Le comportement attendu par le tissage d'aspects peut varier d'une application à une autre. Pour toutes ces raisons, la détection des interférences d'ordre sémantique est une tâche complexe. *Le mécanisme que nous proposons s'intéresse donc à la détection des interférences syntaxiques au niveau des points de jonction partagés.*

TABLE 2.1 – Synthèse Détection et résolution des interférences.

	Détection	**Résolution**	**Techniques**	**Extension du mécanisme**
[MMT06]	analyse statique CSP (Critical Pair Analysis)	Précédence statique	Graphes (diagramme de classe)	-
[CHA+10]	Analyse de graphes d'état du système obtenu en faisant varier l'ordre de tissage	-	Analyse de graphe avec Groove	-
[DS02]	observation d'évènements	Opérateurs : Seq, And et Or	FSP (Finite State Process)	Statique
[DBA07]	Analyse de graphes dflow/c-flow	Précédence Statique	Graphe de flot de contrôle	-
[MW09]	Composition des intentions	Précédence	Graphes de flot de données et de flot de contrôle	-
[DMB09]	Points de jonction partagés	Précédence dynamique selon le context	Ordered Meta-Aspect-Manager	-
[Mos10]	Analyse statique de graphes de flot de données et flot de contrôle	Spécifié par le développeur	Logic du premier ordre	Statique
[CFW09]	Analyse d'arbre de flot de contrôle	Fusion des opérateurs	Logique du premier ordre	Statique
[KK08]	Propriété assume/guarantee	-	LTL (Linear Temporal Logic)	-
[DFQJ08]	Propriété assume/guarantee	-	Logique du premier ordre	-
[WTR07]	Analyse de graphes flot de données	-	Graphe de flot de données	-

2.1.3.2 La résolution des interférences

La technique qui sera utilisée pour la résolution d'interférence s'intègre dans le même cadre que celui du mécanisme de la détection. C'est à dire que si nous souhaitons un mécanisme de détection ouvert (qui accepte l'ajout ou le retrait d'aspects), la résolution doit l'être aussi. La résolution des interférences pour un système ouvert nécessite l'application d'une stratégie capable d'évoluer dynamiquement pendant l'exécution du système. Dans la littérature, la stratégie de résolution la plus utilisée est celle de la *précédence*. L'ajout d'un nouvel aspect va être suivi par le besoin de retrouver le nouveau ordre de tissage de ces aspects (manuellement ou bien automatiquement) et certaines approches nécessitent de modifier les aspects pour intégrer ce nouvel ordre. Avoir un ordre de tissage unique qui résout tous les problèmes d'interférence n'est pas toujours un objectif atteignable (exemple des aspects de temps réel et de synchronisation qui s'appliquent aux même actions). Pour certaines aspects, quelque soit l'ordre dans lesquels ils seront tissés, le problème d'interférence persiste. Dans [CFW09] un mécanisme de fusion des aspects a été proposé pour traiter le problème d'interférences. La solution est axée sur la connaissance de la sémantique des opérateurs utilisés dans la spécification des aspects. **Nous adoptons la même logique de résolution mais nous l'adaptons à nos besoins.**

Définir un mécanisme de gestion des interférence qui sera facilement extensible pour supporter des nouveaux besoins est particulièrement important dans notre cadre de travail. La tableau 2.1 montre que peu d'approches supportent l'extensibilité du mécanisme de gestion des interférences et même s'ils le permettent c'est fait de manière statique. Nous envisagerons alors la définition d'un mécanisme de gestion des interférences qui supporte l'extension dynamique.

La fusion comportementale des interactions logicielles dans [Ber01] et la fusion des aspects dans [CFW09] se basent sur les opérateurs fournis par les langages utilisés. Les opérateurs au niveau du langage peuvent être vus comme des connecteurs au niveau de l'architecture logicielle (assemblage de composants ou orchestration de services). Pour exploiter cette méthodologie de résolution, nous avons besoin de définir le concept de connecteur ainsi que les techniques utilisées pour leur composition. Nous étudions alors les approches dans la littérature permettant la définition et la composition des connecteurs.

2.2 Les connecteurs : une formalisation des interactions

Cette section vise à étudier une des facettes de la composition des entités logicielles : les *interactions*. La formalisation de plus haut niveau des interactions est le concept de connecteur. Cette étude est menée au travers de différents travaux de recherche. Nous définissons tout d'abord le concept de connecteur et les tâches pouvant être assurées par un connecteur d'une manière générale. Nous montrons que nos connecteurs vont assurer principalement une fonction de *coordination* et de *synchronisation*. Par la suite, nous étudions deux modèles de connecteurs représentatifs de cette catégorie. Ces modèles définissent des connecteurs primitifs avec des mécanismes permettant leur composition pour construire des connecteurs complexes. Nous rappelons que la composition de connecteurs est la méthodologie de résolution qui va être adoptée par notre mécanisme de gestion des interférences. Donc, nous nous intéressons particulièrement aux mécanismes de composition des connecteurs.

2.2.1 Définitions

Dans la littérature plusieurs termes sont utilisés pour désigner les relations entres les entités logicielles tels que : dépendances, relations et interactions. Notre cadre de travail se caractérise par la présence des entités logicielles s'exécutant dans un environnement dynamique. Pour cette raison, dans la suite nous utilisons le terme *interaction* puisqu'il met le focus sur la coté dynamique d'un comportement et qui se définit dans [Ber01] comme suit :

Définition 4 :
Une interaction est une entité ayant la charge d'interconnecter un ensemble d'objets afin que leurs comportements s'influencent mutuellement.

Dans les approches traditionnelles, les interactions font partie du code fonctionnel des entités logicielles. Dans ce cas, une entité logicielle possède une connaissance d'autres entités avec lesquelles elle va interagir. Ceci limite de manière significative leur réutilisation ainsi que leur indépendance par rapport aux interactions dans lesquelles elles vont être impliquées. Cette connaissance n'est pas possible dans notre cadre d'étude puisque les entités à composer sont découvertes pendant l'exécution de l'application. Pour remédier à ce problème, d'autres approches ont confié les interactions à des entités complémentaires ce qui permettra de dissocier la définition des entités logicielles de celle des interactions. Ceci est rendu possible grâce à la définition explicite des points d'accès qui sont les ports au niveau des entités logicielles. L'abstraction des interactions de plus haut niveau a été définie clairement dans les ADLs (Architecture Description Language) qui proposent l'utilisation de **connecteurs comme une classe de premier ordre**. Les connecteurs correspondent alors à des entités responsables du contrôle de tous les aspects liés à l'interaction entre les entités logicielles à composer. Dans la suite, nous nous intéressons au concept de connecteur tel qu'il est défini par les ADLs.

2.2.1.1 Le modèle de connecteur

La motivation principale du concept de connecteur est la séparation entre la logique de l'interaction et le code fonctionnel des entités logicielles. Contrairement aux entités logicielles qui sont des boîtes noires, un connecteur est une entité d'encapsulation **boîte blanche** qui expose clairement sa sémantique (il implémente un schéma de communication). La définition formelle du concept de connecteur a été fournie dans la plupart des langages ADL tel que UniCon [SDZ96], Wright [All97] et ACME [GMW10].

La spécification formelle d'un connecteur [All97] montre qu'il est composé principalement d'un ensemble de rôles et d'une description (dite aussi *glue*). Les rôles d'un connecteur forment ses points d'accès (équivalent aux ports des entités logicielles) et spécifient le comportement des participants dans l'interaction (par exemple client, serveur, etc.). Nous retrouvons la même distinction entre rôle d'entrée (ou fourni) et rôle de sortie (requis) que celle utilisée pour les entités logicielles. Ces interfaces décrivent les mécanismes de connexion entre les participants dans l'interaction. L'utilisation des rôles offre *un couplage faible* vis-à-vis des entités logicielles interagissant. La définition du connecteur proposé *ne se repose pas sur les entités logicielles qui vont être impliquées dans l'interaction*. Il peut être réutilisé dans diverses interactions. La partie description du connecteur définit le protocole associé à cette interaction. Il pourra être *prédéfini* (par exemple utiliser SOAP, RMI, etc.) ou bien personnalisé, c'est-à-dire spécifié par le concepteur.

Un exemple d'ADL qui définit un ensemble fixe de connecteurs est Unicon[SDZ96]. Les connecteurs qui peuvent être utilisés sont spécifiés dans un langage et ne peuvent pas être modifiés (Pipe, FileIO, Procedure Call, Remote Procedure Call). Ceci a comme avantage de *bien maîtriser la sémantique du connecteur utilisé par tous les développeurs*. Par contre, avoir un nombre limité de connecteurs constitue un obstacle par rapport à l'expressivité des interactions. Le concepteur ne peut pas utiliser un connecteur en dehors de ceux fournis par le langage. Pour remidier à cette limitation, d'autre ADLs autorisent la personnalisation du protocole implémenté par le connecteur ce qui permettra de définir par exemple la coordination entre les différents participants dans l'interaction.

Dans Wright [All97], la glue d'un connecteur est décrite à l'aide d'un langage particulier CSP (Communicating Sequential Process) [Hoa78]. Ce langage est basé sur une logique de composition d'actions [Hoa69] qui décrit en détail le processus de communication entre les entités impliquées. Par exemple, dans un connecteur de type « *Appel de Procédure* », nous pouvons indiquer à l'aide de ce langage que l'appelant doit initialiser l'appel et que l'appelé doit retourner une réponse. Un connecteur peut aussi avoir des propriétés qui concernent tout ce qui n'est pas lié à sa sémantique. Dans le langage ACME [GMW10], le connecteur précise des propriétés non fonctionnelles telles que son mode de communication, la cardinalité d'un rôle, etc. *Ce modèle montre bien qu'un connecteur est une entité d'encapsulation boîte blanche qui expose explicitement sa sémantique et fournir explicitement son modèle comportemental*.

Pour mieux comprendre le concept de connecteur, il est utile d'identifier et d'analyser les tâches de base qu'un connecteur doit effectuer.

2.2.1.2 Les tâches d'un connecteur

Pour définir les tâches assurées par un connecteur, nous utilisons la taxonomie des connecteurs présentés dans [MMP00]. À partir des types de connecteurs de base présentés, nous avons sélectionné les tâches de connecteurs que sont généralement considérées comme indispensables.

Le contrôle et la transmission de données ($\mathcal{T}_1$) Un connecteur précise les mécanismes sur lesquels sont basés le contrôle et/ou le transfert de données (comme l'appel de procédure, la gestion des événements et flux de données). Chacun de ces mécanismes a des caractéristiques et des propriétés spécifiques ; par exemple, un appel de procédure peut être local ou distant. De même, la gestion des événements peut être basée sur un canal d'événements, une file d'attente d'événements centralisée, etc.

Adaptation d'interface et conversion de données ($\mathcal{T}_2$) Face à la nécessité de lier deux ou plusieurs entités logicielles qui n'ont pas été conçues à l'origine pour inter-opérer, une idée simple est d'inclure un adaptateur dans l'abstraction du connecteur. Pour accomplir cette tâche, le défi à résoudre est celui de mettre en œvre un mécanisme pour la génération automatique des adaptateurs et/ou des convertisseurs de données. Ceci nécessite une connaissance à priori des types des conversions à faire et par conséquent une connaissance partielle des entités qui vont utiliser ces connecteurs.

Coordination d'accès et la synchronisation ($\mathcal{T}_3$) Chaque entité logicielle, utilisée pour la construction d'un système, offre des fonctionnalités à travers ses ports. Pour construire de nouvelles fonctionnalités en se basant sur celles offertes, il faut utilisé des connecteurs qui ajoutent une logique de

coordination (par exemple des actions peuvent être réalisées en parallèle) et qui créent des points de *synchronisation* (par exemple il faut attendre la fin de deux tâches pour commencer une troisième).

Intercepter les communications ($\mathcal{T}_4$) Nous avons vu que les connecteurs interviennent dans les interactions entre les entités qui composent un système. Ils fournissent alors un cadre naturel pour intercepter leur communication et intégrer une autre logique à la communication (par exemple filtre, compression de données, cryptage, etc.)

Considérons à titre d'exemple la composition d'un système en utilisant 3 entités boîtes noires : une horloge, un thermomètre et un écran. Ces entités exposent leurs fonctionnalités à travers leurs ports (utilisés pour échanger des données avec leur environnement). L'horloge possède un port de sortie à travers lequel elle produit périodiquement une chaîne de caractères qui représente l'heure actuelle. De même, le thermomètre comporte un port de sortie qui produit périodiquement une chaîne de caractères qui représente la température en cours. L'écran a un port d'entrée par lequel il consomme périodiquement une chaîne de caractères et l'affiche. Nous pouvons construire différents systèmes en faisant varier la composition de ces entités. Nous pouvons créer une interaction entre l'horloge et l'écran (respectivement entre le thermomètre et l'écran) pour afficher l'heure (respectivement la température) périodiquement. Cette interaction consiste à créer un connecteur d'échange de données (par exemple pipe) entre les entités. Nous considérons maintenant un troisième système qui affiche alternativement l'heure et de la température. Nous devrions être en mesure de composer ces entités dans la bonne façon pour obtenir ce comportement. Il est évident que le connecteur utilisé précédemment ne répond pas à ce besoin. Dans ce cas nous avons besoin d'un connecteur qui va coordonner les entités logicielles depuis l'extérieur pour obtenir le comportement alternatif. *Nous avons évoqué dans le chapitre 1 que nos systèmes logiciels vont être construits en réutilisant les fonctionnalités offertes par les entités logicielles disponibles à un instant donné. Dans un tel cas, la composition vise à définir une logique de coordination entre les diverses fonctionnalités disponibles pour obtenir le comportement souhaité. Pour cela, nous avons besoin de décrire à travers une ou plusieurs compositions la logique qui doit exister entre ces différentes entités logicielles. Cette logique de coordination est confiée aux connecteurs*.

$\hookrightarrow$ **À retenir.** Nous utilisons les connecteurs comme une abstraction des interactions lors de la composition des entités logicielles. Les deux tâches de conversion et d'interception des communications forment des connecteurs de sémantique particulière et ne sont pas forcement des tâches effectuées par tous les connecteurs. Par exemple, si nous avons besoin d'une conversion oui bien d'un adaptateur, nous définissons un connecteur qui implémente explicitement ces fonctionnalités. *Par conséquent, nous considérons que tous nos connecteurs doivent accomplir deux tâches principales : le contrôle et la transmission des données* ($\mathcal{T}_1$) *et aussi la coordination et la synchronisation* ($\mathcal{T}_3$).

La définition des tâches principales assurées par les connecteurs fait partie de la spécification de l'ensemble de connecteurs de base fourni par le modèle de connecteur. Cependant, *"A model wherein interaction is a first-class concept must provide two things : first, primitive interactions ; and second, rules of composition for combining interactions into more complex ones"*[A$^+$05]. Cette définition montre qu'un modèle de connecteur devra définir aussi les règles de composition pour ses connecteurs primitifs. Nous détaillons dans les deux sections suivantes des approches pour la définitions des connecteurs primitifs et la modalité de leur composition.

2.2.2 Connecteurs primitifs (Liaison, canal)

Un connecteur primitif modélise une interaction simple qui correspond à un canal de communication et possède généralement une entrée et une sortie. Plusieurs terminologies ont été utilisées pour désigner un connecteur simple. Par exemple, il est appelé *liaison* dans Fractal [BCL$^+$06] (binding en anglais) et *canal* dans le modèle Reo [AM02] (channel en anglais).

Un connecteur pimitif spécifie des moyens d'interactions différents qui peuvent être classés en deux grandes familles [MCE02] : *les interactions synchrones* et *les interactions asynchrones*. Le modèle d'interaction synchrone impose un couplage fort entre les entités en communication : l'émetteur n'envoie un message que si et seulement si le destinataire est disponible pour recevoir ce message. De ce fait, l'émetteur reste bloqué en attendant cette disponibilité. Les interactions asynchrones offrent un découplage fort entre les entités sous plusieurs formes : requête-réponse asynchrone (l'émetteur de la requête ne reste pas bloqué), communication événementielle et publication-souscription (la souscription d'une entité à un événement publié par une autre créé éventuellement un canal de communication entre les deux). L'avantage de ce dernier patron d'interaction (asynchrone) est le fort découplage entre les entités qui n'ont pas besoin de se connaître par avance.

La nature des informations fournies par la description comportementale d'un connecteur primitif dépend du modèle qu'il implémente [Arb98]. Un connecteur peut être décrit selon le modèle flot de données ou bien flot de contrôle qui se définissent comme suit :

Modèle flot de données L'activité dans une application flot de données est centrée autour des données qui sont partagées et la préoccupation principale est ce qui se passe pour ces données. Les exemples de ce type d'application incluent les systèmes de bases de données et de transactions (les banques, les compagnies aériennes, etc.). Un exemple de langage de coordination qui utilise le modèle flot de donnée est Linda [Gel85]. Le raisonnement se focalise sur la disponibilité des données et leurs transformations. Les instructions dans ce modèle sont non ordonnées et peuvent être exécutées dès que leurs opérandes sont disponibles.

Modèle flot de contrôle Le flux de contrôle se réfère à l'ordre dans lequel les instructions ou des appels de fonction d'un programme sont exécutés ou évaluées. Les types d'opération de flux de contrôle prises en charge par les différentes langues varient. Les opérations les plus répandues sont : les boucles, le branchement conditionnel, arrêt inconditionnel, etc. Dans ce modèle, les activités consomment réellement leurs données d'entrée, et produisent par la suite des nouvelles données qu'elles génèrent par elles-mêmes. Le raisonnement porte sur l'ordre dans lequel les activités doivent être accomplies et non plus sur ce qui se passe sur les données.

Les entités logicielles telles que définies dans notre cadre de travail sont des boîtes noires accessibles à travers des ports (ε_2). Chaque entité consomme alors des données à partir des ports d'entrées et produit de nouvelles données qui seront transmises à travers leurs ports de sortie. Les données sont alors masquées par ces entités et il n'y a pas une trace qui peut être obtenue pour ces données. Dès qu'une entité consomme des données, et même s'il n'y a pas eu une transformation de ces données, nous considérons qu'il s'agit d'une nouvelle donnée qui a été produite par cette entité logicielle (dont nous ne connaissons pas le comportement interne). *Pour cette raison nous représentons le système selon le modèle flot de contrôle. Nous nous intéressons alors dans la suite aux connecteurs de coordinations qui implémentent des chemins de flot de contrôle*.

Dans les deux sections suivantes, nous détaillons deux modèles représentatifs de connecteur de

coordination de type flot de contrôle qui ont été appliqués pour la composition des entités logicielles composant et service.

2.2.2.1 Le modèle du connecteur Reo

Reo [A+05] est un langage de coordination conçu par l'équipe de recherche SEN3 au CWI (Centre for Mathematics and Computer Science) en 2001. Il est conçu pour la gestion des interactions complexes et dynamiques entre des entités indépendantes qui coopèrent dans un système concurrent. Il définit les interactions comme suit : *"An interaction is a constraint : an explicit relation that holds among a set of actors, and constrains each to coordinate their collective behavior"*[A+05]. Cette définition exprime explicitement que les interactions entre les entités est une *coordination exogène*.

Ce modèle présente un paradigme pour la composition des entités logicielles (service et composant) basés sur la notion de canaux (*channel*). Un canal possède *n* sources et *p* destinations. À chacune de deux extrémités est associée un nœud. Il y a trois types de nœud : *source*, *destination* et *mixte* (il est source pour un canal et destination pour un autre). Divers connecteurs de base [MSA06] ont été définis formellement dans Reo et peuvent être groupés dans deux classes : *connecteur synchrone* et *connecteur asynchrone*. Nous classons les connecteurs selon le nombre de leurs ports d'entrées et de sorties (sous la forme *cardinalité entrées - cardinalité sortie*). Par exemple, la famille de connecteur synchrone est composé de :

Connecteurs sous la forme 1-1 Trois exemples ont été définis comme des connecteurs primitifs dans cette catégorie :
- *Synchronous connector* possède un nœud source (port) et un nœud destination et permet de synchroniser le transfert de données entre les deux. L'émetteur (respectivement le destinataire) reste bloqué en attendant la disponibilité du nœud destinataire (respectivement émetteur).
- *Synchronous drain connector* lit les données à partir de ses deux nœuds sources. Il n'y a pas de nœud destination donc les données lues seront perdues.
- *Synchronous lossy connector* possède un nœud source et un autre un nœud destination et permet de synchroniser la destination avec la source et pas inversement. C'est-à-dire que si le destinataire n'est pas disponible pour une raison ou autre, la source ne reste pas bloquée et libère les données sur le canal et par conséquent celles-ci vont être perdues.

Connecteurs sous la forme 1-n Le connecteur primitif qui appartient à cette catégorie est *Fork*. Ce connecteur possède un seul nœud source et plusieurs nœuds destination. Il duplique les données fournies par la source vers toutes les destinations.

Connecteurs sous la forme n-1 le connecteur proposé est le *Merge*. Il possède plusieurs nœuds source et un seul nœud destination et permet de choisir une donnée fournie par l'une des sources (d'une manière aléatoire) pour être transférée vers la destination.

Ce modèle de connecteur n'est pas lié à un paradigme particulier. Il a été utilisé comme un langage pour la construction de connecteurs qui orchestrent les comportements coopératifs des instances de composants (dans un système à base de composants [Arb03]) et aussi l'orchestration de services (d'une application orientée service [LA07]). Ce modèle a défini des connecteurs en reprenant la plupart des patrons utilisés pour l'orchestration de service (une description détaillé est disponible dans [CWI13]). Reo a apporté une contribution originale aux modèles de connecteurs. La modélisation de connecteurs à l'aide d'automates a permis d'exprimer le comportement visé pour

chaque connecteur ainsi qu'un moyen pour une validation formelle [MSA06]. Nous verrons plus tard que ces connecteurs peuvent être composés pour construire un connecteur plus complexe.

Le connecteur présente un moyen de coordination anonyme *c'est à dire que les entités engagées dans l'interaction ne doivent pas se connaître à priori. Il nous semble primordial de rendre explicite le comportement de nos connecteurs pour mieux maîtriser le comportement souhaité par une composition d'entités logicielles et pour pouvoir les valider facilement. L'utilisation du modèle d'automate dans [MSA06] a permis de maîtriser la sémantique des interactions et de vérifier des propriétés sur le système composé.*

2.2.2.2 Autre modèle de connecteur

Lau et al [LEW05] introduisent un modèle de connecteur assez original dont le but est d'encapsuler dans une entité de premier ordre (le connecteur exogène) le détail de contrôle lors de la composition des entités logicielles. Le modèle proposé vise une indépendance complète entre les entités logicielles à composer. Pour cette raison, l'initiation du contrôle a été confiée au connecteur au lieu de l'entité elle-même.

Selon ce modèle, un connecteur de coordination gère l'invocation des services offerts par les entités logicielles ainsi que leur résultat de retour. Ils sont utilisés pour déterminer le flot de contrôle dans un système. Deux catégories de connecteurs ont été proposées : des connecteurs pour l'adaptation et d'autres pour la composition.

Un connecteur d'adaptation est de type *unaire*. Il est utilisé pour contrôler depuis l'extérieur l'appel d'une fonctionnalité d'une entité (il exécute une action de contrôle avant tout appel du port tel que par exemple vérifier une condition externe avant l'appel). Il modifie donc le flot de contrôle.

La deuxième classe de connecteurs proposée dans ce modèle est celle de connecteur de composition et est utilisée pour définir des relations n-aire entre les entités logicielles. Principalement, trois connecteurs de base ont été définis : *Selector*, *Sequencer* et *Pipe* (le catalogue de connecteur est détaillé dans [VEL10]). Les connecteurs Sequencer et Pipe peuvent être utilisés pour composer deux ou plusieurs entités de sorte que l'exécution de chacune d'entre elles est effectuée dans un ordre séquentiel. Le Pipe encapsule aussi le flot de données entre les entités qui participent à ce connecteur. Par exemple $Pipe(e1, e2)$ implique l'exécution de l'entité $e1$ en premier lieu et ensuite faire passer le output de $e1$ comme input à l'entité $e2$ pour pouvoir s'exécuter. Le connecteur de composition *Selector*, correspond à la structure de contrôle de branchement. Ainsi, il peut être utilisé pour composer un ensemble de deux ou plusieurs entités de sorte que l'exécution d'une seule d'entre elles est effectuée, la sélection étant basée sur l'évaluation d'une expression booléenne encapsulée dans le connecteur. Les schémas de contrôle encapsulés dans ce modèle de connecteur sont similaires à celui de structure de contrôle définit dans la majorité des langages de programmation (behavioral pattern).

Le schéma de flot de contrôle n'est pas le même par rapport au modèle Reo. La figure 2.6 montre une composition d'entités logicielles selon les deux modèles présentés.

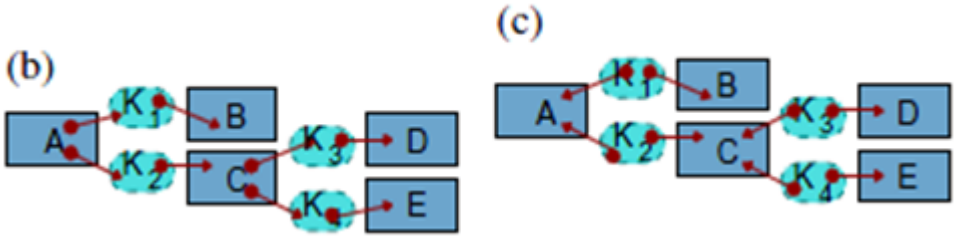


FIGURE 2.6 – Le flot de contrôle selon Reo (b) et selon Lau (c)

Dans Reo le contrôle est déclenché par l'entité logicielle A (dans la figure 2.6.b il y a une flèche qui part de A vers $K1$ et $K2$) et passe par la suite aux connecteurs $K1$ et $K2$ qui à leurs tours vont le passer aux autres entités B et C. Dans l'autre modèle, le contrôle est déclenché par les connecteurs $K1$, $K2$, $K3$ et $K4$ (dans la figure 2.6.c, le sens des flèches est des connecteurs vers les entités logicielles). À partir de cette représentation, nous ne pouvons pas faire apparaître la logique de coordination entre les entités logicielles. Pour ce faire, ce modèle propose plusieurs niveaux de hiérarchie entre les connecteurs. La figure 2.7 montre un exemple d'utilisation de connecteurs de composition pour la construction d'un système logiciel. Cet exemple montre l'aspect hiérarchique. Le connecteur Sequencer $SQ2$ appartient au niveau le plus haut de la hiérarchie et implique d'autres connecteurs de niveau plus bas (Pipe niveau 2 et Selector SEL de niveau 1). Le contrôle est alors initié par le connecteur de plus haut niveau dans la hiérarchie.

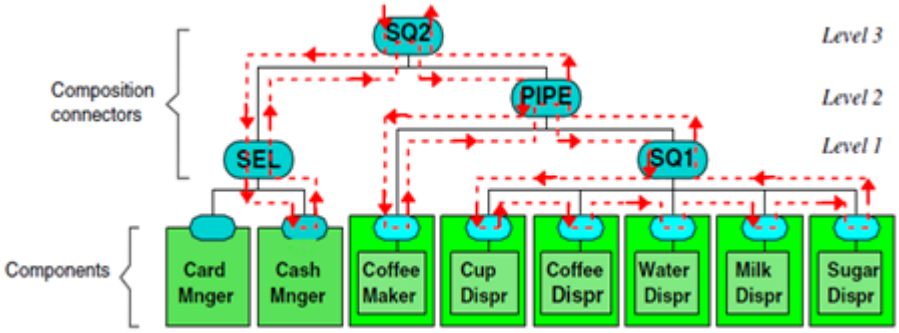


FIGURE 2.7 – Exemple de composition hiérarchique selon le modèle de connecteur de [LEW05]

La délégation complète du contrôle aux connecteurs constitue un inconvénient par rapport à notre démarche car nos entités logicielles s'exécutent dans un environnement dynamique et elles réagissent aux stimuli venant de leur environnement donc l'initiation du contrôle vient forcement d'elles.

L'apparition du concept connecteur a été motivée par la séparation de la préoccupation de coordination pour réduire le couplage entre les entités logicielles et augmenter leur réutilisabilité. Quelque soit le niveau d'expressivité fourni par un modèle de connecteurs à travers un ensemble de connecteurs de base, il n'est pas possible d'exprimer toutes les logiques de coordination dont nous aurons besoin dans nos systèmes. Selon [A+05], le modèle de connecteur doit exprimer la façon de composer ces connecteurs de base : *"interactions can be composed together in various ways to yield more*

complex constraints (or, interaction protocols), without any reference to the action sequences or the states of actors". Cela augmente encore le degré de réutilisation puisqu'un connecteur peut également être réutilisé pour exprimer des fonctions plus évoluées. La section suivante détaille des approches de composition de connecteur primitifs pour construire des connecteurs plus complexes.

2.2.3 Composition de connecteurs : le connecteur complexe

Un connecteur complexe (dit aussi composite) peut être obtenu par composition de connecteurs primitifs. Il existe des modèles de connecteurs qui autorisent l'intégration d'une entité logicielle dans l'architecture interne d'un connecteur. Cependant, dans notre cadre d'étude, les entités logicielles sont des boîtes noires. Leur intégration dans l'architecture interne d'un connecteur réduit la maîtrise de sa sémantique. Nous excluons donc cette catégorie de travaux de notre étude et considérons qu'un connecteur fournit une vision boîte blanche avec une maîtrise complète de son comportement. De nombreuses approches existent pour définir de tels mécanismes de composition. Nous détaillons deux catégories pour la composition de connecteurs : des modèles de *composition hiérarchique* et des modèles de *composition comportementale*.

2.2.3.1 Composition hiérarchique

Un modèle de composition de connecteurs est considéré comme étant hiérarchique si la définition d'un connecteur complexe se fait en regroupant un ensemble de connecteurs de base sous un nouveau nom. Nous proposons d'étudier deux modèles représentatifs de ce type de composition.

Lopes et al. [LWF03] ont proposé une formalisation de la notion de connecteur complexe. Un connecteur simple est conforme au modèle défini dans la plupart des ADLs, c'est à dire, un ensemble de rôles et une partie glue. La contribution originale de leur travail est qu'un connecteur complexe encapsule un ensemble de connecteurs qui forme sa partie glue. Pour illustrer leur modèle, nous présentons un exemple simple d'un connecteur *C* qui a deux rôles : *expéditeur* et *récepteur*. L'entité logicielle *A* expédie un message à une autre entité *B* via le connecteur *C*. Supposons que nous souhaitions ajouter une compression au transfert de ce message. *Lopes et al.* proposent que le nouveau connecteur *D* se construise par une composition du connecteur *C*. *D* aurait deux rôles : *compression* et *décompression*. La partie glue de nouveau connecteur *D* sera formulée en utilisant celle du connecteur *C*. Ceci est illustré par la figure 2.8.

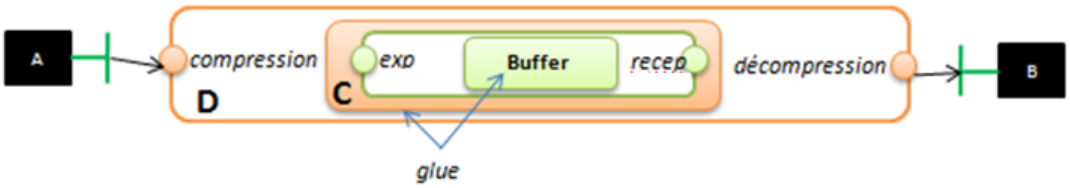


FIGURE 2.8 – Composition de connecteurs par encapsulation de connecteurs primitifs

Un autre modèle de composition de connecteur a été défini dans [LUVW06]. *Lau et al.* proposent la composition de connecteurs dans un modèle hiérarchique. Les connecteurs composites sont des mécanismes complexes de connexion spécialement conçus pour faciliter la communication entre les

composants d'un système en regroupant plusieurs connecteurs de base et/ou complexes. Des connecteurs basiques peuvent être encapsulés dans une nouvelle entité et être considérés comme un nouveau connecteur complexe (figure 2.9). L'interface du connecteur complexe est l'union des interfaces de ses constituants et son implémentation fait appel à l'implémentation de ses sous-connecteurs. Dans un tel modèle de composition hiérarchique, le connecteur complexe réduit les niveaux de complexité dans l'architecture du système de sorte que la composition est plus efficace. Lau et al proposent un catalogue de connecteurs complexes [LLUE07] tel que *exclusive choice sequencer*, *simple merge sequencer*, etc. Par exemple le connecteur complexe *Observer* de la figure 2.9 encapsule les deux connecteurs *PipeetSequencer*.

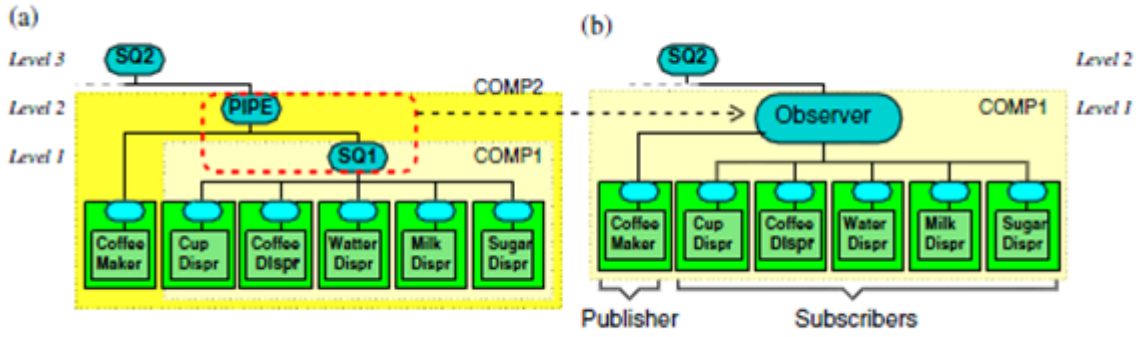


FIGURE 2.9 – Connecteur complexe obtenu par un regroupement de connecteurs de base

La contribution de ce modèle de connecteur complexe est que le comportement du connecteur complexe reproduit les comportements de chacun de ses connecteurs internes en respectant leur niveau de hiérarchie. La figure 2.9.(a) montrent que dans le cas où le Pipe invoque l'entité logicielle $CoffeeMaker$, il reçoit en retour des données et par la suite va les transmettre au connecteur SEQ. Ce dernier invoque toutes les entités logicielles de type Dispenser de sorte que chaque composant utilise une donnée de pipe. La figure 2.9.(b) reproduit ce comportement en utilisant un connecteur complexe Observer qui est de type Publish-Subscribe. Lorsque le Publisher est invoqué ($CoffeeMaker$), les abonnés (Subscribers) doivent être notifiés et doivent s'exécuter les uns après les autres.

La limitation des approches proposées est que la définition du connecteur complexe doit être effectuée pendant la phase de conception du système. Cependant, nous avons déjà évoqué que dans notre approche nous allons procéder à la composition des connecteurs pour résoudre les problèmes d'interférences. Donc cette composition devra s'effectuer durant l'exécution du système sans l'arrêter. Nous avons besoin d'une approche de composition dynamique de connecteurs.

Les approches de composition dynamique de connecteurs sont axées sur la définition formelle du comportement des connecteurs de base. Nous présentons dans la sections suivante deux approches pour la composition comportementale.

2.2.3.2 Composition comportementale

La composition comportementale des connecteurs propose de combiner les comportements fournis par chaque connecteur de base pour fournir un nouveau connecteur. Cette composition exige une description formelle du comportement des connecteurs de base. L'approche proposée sera alors fortement liée au modèle formel utilisé.

Spitznagel et Garlan [SG03] ont défini un cadre pour la composition de connecteurs. Dans ce modèle, un connecteur est défini par un six-uplet comprenant : son code, des bibliothèques de communication, des services d'infrastructure (système d'exploitation), des données (localisation des participants à la communication), des politiques (par exemple l'ordre d'appel) et un comportement formel. Dans ce modèle, les connecteurs sont composés en utilisant un ensemble de transformations qui sont des opérateurs applicables à une ou plusieurs parties d'un connecteur. Spitznagel et Garlan ont démontré leur approche en combinant des connecteurs en utilisant un ensemble de transformations de base. Par exemple, ils proposent comme opérateur de transformation : *Aggregate* et *Splice*. Aggregate est un contrôleur qui active un seul connecteur à un instant donné. L'opérateur Splice permet quant à lui de *combiner le code binaire de deux connecteurs* c_1 et c_2 dans un nouveau connecteur c_3. Ce dernier possède une interface de c_1 et une autre de c_2. La contribution principale de ce travail est la définition d'un ensemble d'opérateurs de transformation qui peuvent être utilisés pour composer des connecteurs primitifs et obtenir un connecteur complexe. Cette composition utilise la description du modèle comportemental des connecteurs. L'approche proposée est fortement liée à leur propre modèle de connecteur. Elle est difficilement exploitable pour l'appliquer à d'autres modèles. Cet obstacle peut être résolu en s'appuyant sur une représentation générique du modèle de connecteur.

Une représentation générique du modèle de connecteur a été proposé dans [BSAR06]. La composition dynamique des connecteurs primitifs pour la construction d'un connecteur complexe a été une contribution originale du modèle Reo [BSAR06]. Il propose un modèle formel pour la construction d'un connecteur complexe [SFBK$^+$12]. Dans ce modèle, la composition de connecteurs primitifs est une composition des automates correspondant à leur comportement. La composition des connecteurs ne se limite pas au simple échange d'événements entre les automates de chacun d'entre eux mais plutôt permet la définition d'un nouvel automate qui inclut les contraintes de synchronisation exprimées par tous les automates des connecteurs à composer. Ce modèle propose donc *la fusion comportementale des connecteurs de base pour construire un nouveau connecteur complexe*. Une contribution originale à ce modèle est l'introduction d'un mécanisme de reconfiguration des connecteurs. Ce mécanisme repose sur un ensemble de règles de transformation de graphes [Tae00]. Un connecteur peut être ajouté ou bien supprimé pendant l'exécution de la même manière que les entités logicielles. Ceci est un critère très important car il permettra au connecteur de suivre la dynamique de l'environnement. Les règles de reconfiguration permettant la modification des connecteurs peuvent être contradictoires les unes par rapport aux autres. Pour cela un mécanisme pour détecter ces éventuels problèmes a été utilisé. Mais il manque la définition d'une solution aux problèmes identifiés.

La composition de connecteurs proposée par ce modèle présente plusieurs avantages : (1) La composition de connecteurs primitifs se fait à partir de leur modèle formel d'automates, ceci permet de s'assurer de la validité du comportement du connecteur complexe, (2) L'utilisation d'une représentation abstraite du système à l'aide des graphes pour être indépendant de l'implémentation et (3) La composition des connecteurs est dynamique et pourra être modifiées en utilisant un ensemble de règles de transformation de graphes.

2.3 Synthèse et objectifs

L'étude de l'état de l'art nous a permis de tirer des caractéristiques qu'un mécanisme de composition devra fournir tout en respectant les contraintes imposées par notre cadre d'étude. Dans la suite nous définissons ces caractéristiques.

Projection dans plusieurs modèles : Le mécanisme de composition ne doit pas se limiter à un modèle d'implémentation particulier et devra s'appliquer tel qu'il est (sans besoin d'être adapté) pour la composition des diverses entités logicielles. Ces entités logicielles peuvent être projetées par la suite dans plusieurs modèles sous-jacents : composant, service, service pour dispositif etc.

Composition dynamique : La composition peut avoir lieu pendant l'exécution de l'application pour y intégrer des nouvelles fonctionnalités (suite à l'apparition des nouvelles entités logicielles dans l'environnement). Les mécanismes de composition qui interviennent pendant la phase de conception du système ne sont pas donc envisageables.

Coordination exogène : Divers mécanismes de composition ont été proposés dans la littérature. Dans notre cadre d'étude la composition des entités logicielles a comme objectif de définir la logique de leur coordination ($\mathcal{T}_3$) pour que ces dernières arrivent à coopérer et proposer des nouvelles fonctionnalités à l'issue de leur composition. Cette logique de coordination est exogène (externe), c'est à dire qu'elle n'a pas connaissance de la logique interne des entités logicielles (elle sont des boîtes noires ε_2). La logique de coordination est confiée alors à des entités de premier ordre qui sont les connecteurs. De cette manière, ils peuvent être réutiliser pour (1) construire diverses compositions et (2) valider le comportement souhaité par la composition en s'appuyant sur la sémantique des connecteurs.

Indépendance de spécification des entités d'adaptation et Extensibilité : Les entités de de composition doivent être décrites *indépendamment* sans supposer l'existence ou bien l'absence des autres (ε_4). Le mécanisme de composition devra aussi permettre d'ajouter et de retirer à l'exécution des entités d'adaptation (σ_2) (la composition doit est recalculée automatiquement).

Identification et Résolution automatique des problèmes d'interférence : à l'issue de cette indépendance de spécification des interférences peuvent apparaître ; ceci perturbe le résultat de la composition. Donc un mécanisme de composition doit forcément proposer des moyens pour détecter et résoudre ces interférences. Plusieurs critères sont associés à la gestion des interférences qui sont :

- Composition basée sur un modèle à l'exécution : travailler sur une représentation abstraite du système permettra d'être indépendant du détail d'implémentation et par conséquent de disposer d'un mécanisme de gestion des interférences plus générique. Puisque la composition doit intervenir pendant l'exécution, le modèle du système devra aussi représenter l'application qui est en cours d'exécution. L'avantage de travailler au niveau modèle permettra de s'assurer du résultat de la composition avant de le concrétiser au niveau du système qui s'exécute et de garantir un certain nombre de propriétés.
- **Détection** : *Interférence syntaxique au niveau des points de jonctions partagés*. Nous avons vu que la résolution des interférences sémantiques nécessite la connaissance de la sémantique des entités logicielles (ε_2). Nous nous intéressons à la résolution des interférences d'ordre syntaxique qui se produisent au niveau de points de jonction partagés. Nous adressons le problème de **concurrence** entre les entités d'adaptation. Cela signifie que quel que soit l'ordre dans lequel elles sont appliquées, il y a toujours un problème d'interférence (les problèmes de conflit ou de dépendance sont résolus par trouver un ordre totale entre les entités d'adaptation).

- Résolution **Interne** : La résolution est basée sur la fusion des comportements spécifiés par chacune des entités d'adaptation. Le résultat de cette fusion reprend tous les comportements en entrée tout en garantissant la cohérence du comportement résultant. Nous combinons les concepts utilisés pour la composition des connecteurs ainsi que le travaux sur la fusion des opérateurs dans la définition de notre approche de résolution. **Nous proposons la composition des connecteurs dont l'objectif est la résolution des interférences**.
- Résolution **Déterministe** : Le mécanisme de résolution doit garantir le déterminisme du résultat.
- **L'extension dynamique du mécanisme de résolution des interférences**. **Notre mécanisme devra autoriser l'utilisation d'un connecteur non connu à priori**. Notre mécanisme sera étendu automatiquement pendant l'exécution et sera par la suite capable de résoudre des interférences engendrées par l'utilisation de ces nouveaux connecteurs.

Dans cette thèse nous proposons un mécanisme générique pour la détection ainsi que la résolution des interférences d'ordre syntaxique. Dans notre approche le mécanisme, de composition est présenté selon deux niveaux. La phase de conception du mécanisme de gestion des interférence permet (1) de définir au niveau du modèle de graphes les types de problèmes adressés, (2) de proposer un catalogue des connecteurs de composition sous forme de patrons décrivant la structure abstraite de chaque connecteur et (3) de définir un ensemble de règles qui seront utilisées pour la résolution des problèmes identifiés. La phase de conception a pour objectif de valider le mécanisme de gestion d'interférences et de s'assurer que le résultat proposé est cohérent par rapport aux propriétés que le système doit garantir. Dans un deuxième temps, à l'exécution, nous présentons le détail du processus de gestion des interférences. **Notre mécanisme de gestion des interférences ne se limitera pas à un ensemble figé de connecteurs. Il supporte l'utilisation de connecteurs non connus à priori. Ceci est rendu possible grâce à l'intégration d'un mécanisme permettant l'extension dynamique et automatique de la base de règles utilisée pour la gestion des interférences.**

Deuxième partie

Contribution

CHAPITRE 3

La gestion des interférences : un mécanisme extensible

Sommaire

L'objectif de ce chapitre est de proposer un mécanisme de gestion des interférences permettant de respecter les multiples contraintes imposées par le cadre de l'IAm. Dans l'état de l'art, nous avons mis en avant l'importance d'avoir un mécanisme de gestion des interférences se basant sur une représentation abstraite de l'application qui est en cours d'exécution. Dans ce chapitre, nous abordons la spécification d'un cadre formel pour la détection et la résolution des problèmes d'interférences. Nous considérons qu'un système (orienté service ou à base de composants) est décrit par un graphe. Ce graphe est formé par deux classes de nœuds : *boîte noire* et *boîte blanche*. Pour rappel, nous avons vu que les nœuds boîtes noires sont les entités logicielles qui composent un système ambiant (ε_2). La composition de ces entités fait appel à un ensemble de connecteurs de coordination (2.2.2). Ces connecteurs sont connus a priori et forment les nœuds boîtes blanches. Toute la logique de notre mécanisme de résolution se base sur cette connaissance.

Dans ce chapitre, nous commencerons par une description du modèle de graphe qui nous permettra d'intégrer les différents paramètres nécessaires pour notre raisonnement. Nous verrons ensuite les types de problèmes d'interférence que nous adressons dans notre contribution. Après avoir identifié les problèmes, nous détaillerons les différentes classes de mécanisme de résolution et nous soulignerons leurs limitations en terme de prise en compte des connecteurs non connus à l'avance. Nous proposerons par la suite, une évolution de ces mécanismes de gestion des interférences en y intégrant la possibilité d'extension automatique et dynamique.

3.1 Modélisation abstraite des applications

Notre contribution est définie en terme d'une représentation abstraite des applications ce qui permettra de l'appliquer à des systèmes logiciels basés sur des technologies différentes. Dans le chapitre précédent, nous avons souligné les insuffisances relatives aux approches classiques ainsi

que celles utilisant des arbres de flot de contrôle pour résoudre les interférences apparues lors de la composition d'un système. Dans notre travail, nous optons pour l'utilisation des graphes qui offrent un cadre formel évolué et une représentation visuelle facilement compréhensible sans nécessiter de pré-requis. Nous retrouvons dans la littérature plusieurs travaux axés sur une représentation des systèmes par les graphes [WJ08] qui a prouvée un grand intérêt (preuve de consistance [MMT06], vérification de propriétés [CHA+10], etc.).

Un même système pourra être représenté par plusieurs types de graphes dont chacun représente une vue du système (communication, dépendance, composition, etc.). Dans [WJ08] par exemple, le formalise du graphe a été utilisé pour représenter le diagramme de classe de l'application. Dans ce modèle, les nœuds du graphe représentent les classes et les liens définissant leurs relations de dépendances. Nous proposons d'utiliser les graphes pour représenter une application qui est en cours d'exécution. La définition des nœuds ainsi que des arcs dans ce graphe va être différente des approches présentées dans notre état de l'art. Pour rappel, la section 1.3.1 du chapitre 1 a montré que nos systèmes sont composés d'entités logicielles (pas de distinction entre service et composant). Ces entités sont liées par des interactions. Il peut s'agir des liens de type port à port dans le cas d'une interaction simple (figure 3.1). Une interaction peut être définie entre plusieurs entités logicielles et dans ce cas un connecteur est utilisé comme intermédiaire pour définir la logique de coordination entre elles ($\mathcal{T}_1$ et $\mathcal{T}_3$). Ce type d'interaction est illustré dans la figure 3.1 en utilisant le connecteur $conn1$ pour lier les entités $E1$, $E3$ et $E4$. Les liaisons port à port peuvent prendre différentes formes. Nous modélisons ces liaisons comme des messages à sens unique, envoyés par une entité et reçus par un connecteur ou une autre entité. Dans le cas d'un pattern d'interaction nécessitant une réponse, nous le décomposons en deux messages différents. Dans la suite nous définissons notre modèle de graphe pour la description de nos systèmes logiciels.

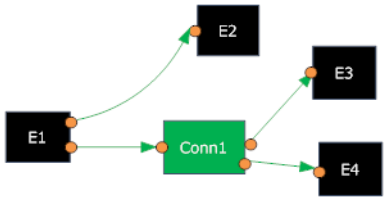


FIGURE 3.1 – Modèle de types graphe de l'application

Un graphe $G(A,N)$ est défini selon le formalisme de base par l'ensemble de ses nœuds N et l'ensemble de ses arcs A dont chacun relie deux nœuds. D'autres systèmes nécessitent une expression beaucoup plus riche au niveau du modèle du graphe. Pour prendre en compte les relations asymétriques entre les nœuds, les graphes *orientés* ont été définis (lorsqu'il existe un arc d'un nœud n_1 vers n_2 alors il n'est pas possible d'avoir un arc de n_2 vers n_1). Dans un tel cas, chaque arc est spécifié par un couple de nœuds (n_1,n_2) où n_1 est le nœud du départ et n_2 est le nœud d'arrivée. Dans un système logiciel, entre les nœuds peut exister des relations différentes. Ces informations peuvent être spécifiées au niveau des arcs en ajoutant des étiquettes. L'utilisation des étiquettes ne se limite pas aux arcs et pourra être utile pour distinguer les nœuds du graphe en leur associant des attributs (nom, type, etc.). Cela forme alors un graphe orienté et étiqueté qui se définit comme suit :

Définition 5 :
Un graphe orienté étiqueté G est un cinq-uplet (N, A, L, E, F_L, F_E) *avec N qui est l'ensemble des nœuds, A est l'ensemble d'arcs orientés, L est l'ensemble des étiquettes associées aux nœuds et E est l'ensemble des étiquettes affectées aux arcs. La fonction* $F_L : N \rightarrow L$ *associe des étiquettes pour un nœud et la fonction* $F_E : A \rightarrow E$ *associe des étiquettes à un arc.*

Dans notre approche, nous utilisons les graphes pour modéliser les flots de contrôle pour un assemblage de composants ou une composition de services. Un graphe est donné à deux niveaux : type (*Typed Graph*) et instance. Au niveau type, nous définissons le modèle de graphe c'est à dire décrire les ensembles *N* et *A*. Nous spécifions aussi au niveau type, les relations entre les différentes entités (cardinalité, contrainte, etc.). Par la suite, toutes les instances de graphe doivent être conformes au *Type Graph* qui a été établi. Plusieurs représentations sont possibles. Par exemple, dans [Gue06] le choix était de représenter les entités logicielles avec des nœuds et leur interactions avec des arcs.

Pour rappel, nous avons vu dans le chapitre 2 que la composition des entités logicielles préexistantes consiste à définir les interactions entre elles en mettant en relation leurs ports. Une interaction simple se produit entre deux ports. La spécification d'une interaction consiste à préciser le port émetteur et le port récepteur. Dans [Fat09] deux types de nœuds ont été spécifiés : des nœuds pour décrire les entités logicielles et les nœuds qui décrivent les ports. Un arc d'une entité logicielle vers un port est utilisé pour associer ce port à cette entité. La figure 3.2 à gauche montre un exemple de cette représentation. L'inconvénient vient ici du faite qu'il faut spécifier trois éléments (deux nœuds et un arc) pour définir un port. Afin de réduire l'ordre du graphe (et donc la complexité de l'approche proposée puisque un graphe plus grand nécessite un traitement plus long), nous utilisons les nœuds pour définir directement les ports des entités ce qui nous permettra de manipuler des graphe d'ordre plus faible. La figure 3.2 à droite présente le même exemple selon notre modèle. L'ordre de graphe dans le premiers cas (à gauche) est 7 alors que dans notre cas il est égal à 4.

Une propriété directement liée à cette représentation est que nos graphes ne sont pas forcement connexes (un graphe est connexe si on peut lier deux nœuds quelconques de ce graphe par une chaîne). Cela est dû au fait que nous représentons par les nœuds les ports des entités logicielle sans expliciter le modèle interne de ces derniers (il n'y a pas de liaison entre les ports d'une même entité logicielle). Cette propriété devra être prise en compte dans la mise en œuvre dans le parcours du graphe.

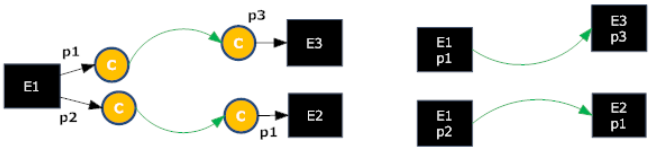


FIGURE 3.2 – Comparaison entre le modèle proposé (à droite) et un le modèle dans [Fat09] (à gauche)

Le Méta modèle de notre *Typed Graph* est donné par la figure 3.3. La classe *Entity* est une abstraction des entités de premier ordre impliquées dans une application. Nous avons évoqué le fait que nous avons deux types d'entités : des entités logicielles prédéfinies de sémantique inconnue appelées

BlackBoxEntity (les ports des services ou des composants) et des connecteurs de sémantique connue *WhiteBoxEntity*.

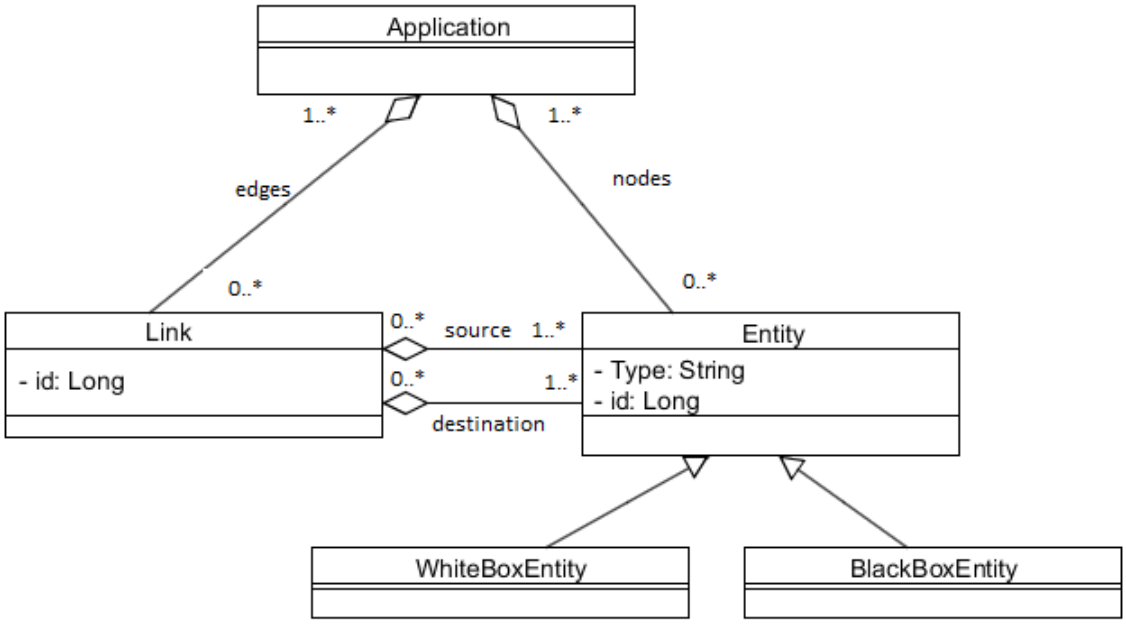


FIGURE 3.3 – Modèle de Types graphe de l'application

En effet, dans une approche boîte noire, les interfaces des entités de l'application sont constituées des ports des composants ou interfaces de service. Dans nos graphes, les nœuds représentent soit les ports des entités logicielles (*BlackBoxEntity*) ou les connecteurs (*WhiteBoxEntity*). Nos graphes sont étiquetés. Chaque nœud est décrit par deux attributs : *id* et *Type*. L'attribut *Type* indique s'il s'agit d'un nœud qui représente un port d'une entité ou bien la sémantique du connecteur. Le deuxième attribut *id* représente l'identifiant du nœud (qui doit être unique). Nous distinguons les ports de sorties (requis) d'une entité logicielle de ses ports d'entrées (fournis). L'approche de résolution des interférences que nous proposons se base sur la connaissance de la sémantique (comportement) des connecteurs utilisés pour décrire le flot de contrôle dans le système.

Dans la suite nous utilisons les notations suivantes :

Notations 5 :
Soit P l'ensemble de ports de toutes les BlackBoxEntity.
L'ensemble P est partitionné en deux sous-ensembles disjoints qui sont les ports de sortie P_s et les ports d'entrée P_e avec $P = P_s \cup P_e$.
Soit $\mathcal{G}$ l'ensemble de connecteurs.
Soit $\mathcal{M}_G$ l'ensemble de graphe conforme à notre modèle.
Soit $n_i \in P_s$, l'ensemble des nœuds successeurs est noté $n_i \bullet [v_i, ..., v_j]$.
Soit $n_j \in P_e$, l'ensemble des nœuds prédécesseurs est noté $[v_i, ..., v_j] \bullet n_j$.

Dans notre modèle de graphe nous avons défini deux classes de nœuds. L'ensemble *P* représente les entités disponibles dans l'environnement avec qui nous ne pouvons qu'interagir sans connaître leur

logique de fonctionnement interne (ε_2). Donc, nous nous intéressons particulièrement aux nœuds de l'ensemble $\mathcal{G}$ qui exposent explicitement leur sémantique. Notre approche de résolution des interférences exploite alors cette connaissance pour formuler la solution.

3.1.1 Les nœuds de sémantique connue

La définition de la sémantique des connecteurs est un point clé dans notre approche puisque toute la logique de résolution que nous proposons est basée sur cette connaissance. Nous avons évoqué précédemment que nos connecteurs accomplissent essentiellement deux tâches qui sont : *le contrôle et la transmission des données* ($\mathcal{T}_1$) et aussi *la coordination et la synchronisation* ($\mathcal{T}_3$). Dans le chapitre 2, nous avons classé les connecteurs proposés dans [AM02] en trois classes. Nous considérons alors ce même classement pour la définition de nos connecteurs : connecteur 1-n, connecteur n-1 et connecteur 1-1. Chaque classe spécifie la cardinalité des arcs d'entrée et des arcs de sortie associés à un connecteur. Nous notons que la définition des interactions de type $n-m$ passe par l'assemblage d'au moins deux connecteurs de type n-1 et 1-m.

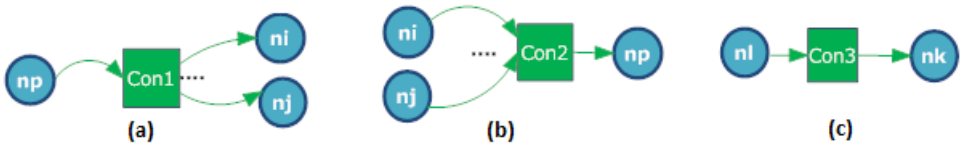


FIGURE 3.4 – Définition de trois classes de connecteurs

Les connecteurs 1-n Dans cette classe, chaque connecteur possède un unique nœud prédécesseurs et plusieurs nœuds (n) successeur comme le montre la figure 3.4.a. Un nœud n_j est un successeur (respectivement prédécesseur) du sommet n_i s'il existe un arc de la forme $e_k = (n_i, n_j)$ (respectivement de la forme $e_k = (n_j, n_i)$). Un connecteur de type 1-n définit un point à partir duquel un seul flot de contrôle peut être divisé en plusieurs flots. Les connecteurs dans cette classe transmettent les données reçues vers les nœuds successeurs. Par exemple, dans cette classe de connecteurs, nous trouvons le connecteur *sequencer* [LEW05] à partir duquel deux flots de contrôle vont être exécutés séquentiellement.

Les connecteurs n-1 Un connecteur est de type n-1 s'il possède plusieurs nœuds prédécesseurs et un seul nœud successeurs (figure 3.4.b). Il définit un point ou plusieurs flots de contrôle peuvent être réunis. Puisqu'il s'agit d'un point qui peut être atteint par plusieurs threads, plusieurs stratégies peuvent être associées à cette classe de connecteurs : *bloquante* ou *non bloquante*. Un connecteur est de type bloquant s'il a besoin de l'activation de tous ces points d'entrées (les prédécesseurs). Un connecteur est considéré comme non bloquant si l'activation de l'une de ses entrées entraîne l'activation de son successeur. Cette classe de connecteurs est utilisée pour synchroniser plusieurs flots de contrôle.

Les connecteurs 1-1 Ce sont les liaisons port à port avec introduction d'un traitement spécifique (par exemple filtrage)

Pour simplifier notre démarche, nous considérons $n = 2$. Par exemple si nous avons besoin de représenter un connecteur où n est plus grand, nous utiliserons plusieurs connecteurs au lieu d'un seul comme le montre la figure 3.5.

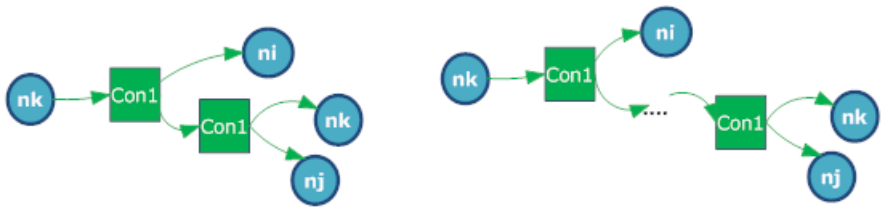


FIGURE 3.5 – (a) Connecteur de type $1-3$ (b) Un connecteur de type $1-n$

Dans cette section, nous nous sommes intéressés à la définition des sémantiques attribuées aux nœuds appartenant à l'ensemble $\mathcal{G}$. Dans la section suivante, nous explicitons la signification attribuée aux liens dans nos graphes.

3.1.2 Sémantiques des liens dans le graphe

Les arcs du graphe modélisent les interactions entre les entités de premier ordre impliquées dans nos applications. Pour un arc (n_i, n_j), n_j est appelé successeur de n_i (respectivement n_i est appelé prédécesseur de n_j). À chaque arc est associé une étiquette qui est fortement liée à la sémantique de son nœud source. Par exemple, pour le connecteur *sequencer*, des étiquettes sont attribuées à ces arcs sortants pour expliciter dans quel ordre les nœuds destination seront activés (c'est le chemin du flot de contrôle).

TABLE 3.1 – Sémantique des liens.

Type :Sequencer	**Étiquettes**
Première opération	$action_1$
Deuxième opération	$action_2$
Opération N	$action_n$

La figure 3.6 est une instance d'un graphe conforme à notre modèle. L'émission de l'événement *SetOn* du composant *Switch* déclenche deux actions : *light1.setTarget* puis *light2.setTarget*.

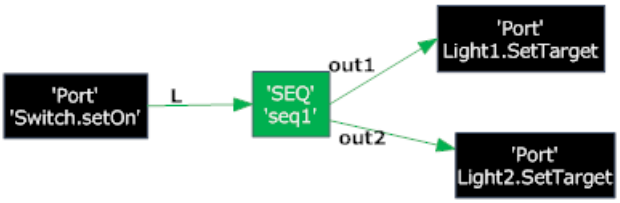


FIGURE 3.6 – Instance d'un graphe selon notre modèle

Nous proposons d'utiliser le modèle de graphe comme une abstraction des systèmes logicielles à l'exécution. Ce modèle sera utilisé par le mécanisme de gestion des interférences pour identifier et résoudre les problèmes apparus pendant la composition. Dans la section suivante, nous présentons les types de problème d'interférence que nous adressons ainsi que la manière dont on détecte ceux-ci.

3.2 La détection des interférences

La gestion des interférences est une étape fondamentale dans le processus de la composition des entités logicielles prédéfinies. Pour rappel, nous nous intéressons à la résolution des interférences syntaxiques au niveau des points de jonction partagés et plus particulièrement aux problèmes de la *concurrence*. Les approches de résolution classiques qui consistent à trouver un ordre d'application des entités d'adaptation sous forme d'aspects ne résolvent pas ce type d'interférence (quel que soit l'ordre, le problème persiste toujours). Le problème de concurrence à détecter est alors fortement lié aux types des points de jonction. Dans notre approche, nos points de jonction sont les ports des entités logicielles (les deux ensembles P_s et P_e).

Le premier type d'interférence (appelé $1-n$) se produit au port de sortie P_s d'une entité logicielle et correspond à un point dans le graphe de flot de contrôle (après l'application de plusieurs squelettes de composition) à partir duquel plusieurs flots de contrôle peuvent potentiellement être activés dans n'importe quel ordre. L'ordre d'exécution de ces actions est variable et peut être différent d'une exécution à une autre. De ce fait, ce point de non-déterminisme peut engendrer des problèmes. Un exemple de ce type d'interférence est illustré dans la figure 3.7 qui présente le résultat de la superposition de deux spécifications de composition au niveau d'un point de jonction partagé $E1.p1$.

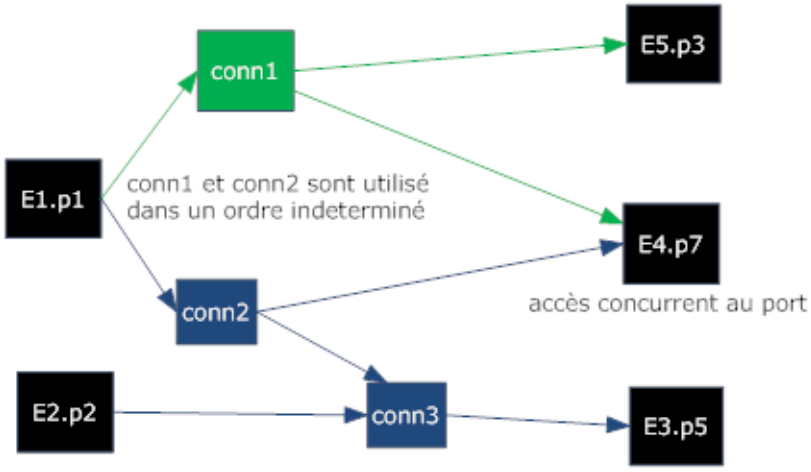


FIGURE 3.7 – Un exemple de deux types d'interférences

Le deuxième type de problème d'interférence (appelé $n-1$) que nous considérons est celui de l'accès concurrent à un port d'entrée P_e d'une entité logicielle (La figure 3.7). Cet accès concurrent est une interférence potentielle car cela pourra être gérer en interne par l'entité logicielle elle-même. Cependant, nous avons évoqué précédemment (section 1.1.2) que ces entités sont des boîtes noires (ε_2) et que nous n'avons aucune connaissance de leur comportement interne. Pour cela, nous considérons que les interférences potentielles sont aussi des interférences et devront être résolues d'une manière externe aux entités qui composent le système.

Au niveau de notre modèle du graphe, les deux types d'interférences seront représentées par deux motifs (patterns). La définition formelle de ces problèmes est la suivante :

Définition 6 : Patrons d'interférence
Soit $g \in \mathcal{M}_G$.
$\forall n_i \in P$ *le nombre d'arcs sortants de* n_i *est noté* $deg^+(n_i)$ *et le nombre d'arcs entrants à ce nœud est noté* $deg^-(n_i)$
Une interférence de type 1-n est définit par : $\exists n_i \in P_s$ *tel que* $deg^+(n_i) \geq 2$
Une interférence de type n-1 est définit par : $\exists n_i \in P_e$ *tel que* $deg^-(n_i) \geq 2$

Dans la figure 3.7 $E1.p1 \in P_e$ et $deg^+(E1.p1) = 2$, il s'agit bien d'un exemple du problème d'interférence de type $1-n$. Un autre problème est détecté au niveau du nœud $E4.p7$ mais cette fois-ci, il s'agit d'un problèmes de type $n-1$.

Cette représentation à l'aide des graphes nous permet d'identifier facilement ces problèmes. Il est possible de détecter les interférences d'une façon entièrement automatisée, en s'appuyant sur les outils existants pour l'analyse de graphes. Lorsqu'un problème d'interférence est détecté, l'endroit où il se situe va être marqué pour mémoriser le lieu où se situe le problème. L'étape de résolution des interférences a pour objectif de produire un graphe dans lequel tous les problèmes ont été résolus. Il s'agit d'une réécriture du graphe résultant de l'étape de détection des interférences.

3.3 Vers un mécanisme Dynamiquement Extensible

La section précédente 3.2 présentait les types de problèmes que nous adressons dans notre travail. La gestion des interférences est un enjeu important pour obtenir un système utilisable. Nous classons les approches de résolution, qui se basent toutes sur la théorie de réécriture de graphe [RE99], en trois catégories. Nous détaillons dans les paragraphes suivants chacune de ces trois catégories.

3.3.1 Résolution classique

Le problème des interférences est dû au fait qu'il y a plusieurs flots de contrôle (relatifs à plusieurs entités d'adaptation) qui sont activés à partir d'un port de sortie d'une entité logicielle ou bien se réunissent au niveau d'un port d'entrées de cette entité. Dans la littérature, plusieurs solutions ont été proposées pour la gestion de ces interférences. Ces approches se basent principalement sur la définition d'un ordre à respecter pendant l'exécution. Nous avons pu démontrer dans le chapitre 2 que, dans notre cadre de travail, cet ordre peut ne pas être existant. Comme dans [CFW09] (fusion des arbres syntaxiques) et [Ber01] (fusion des interactions logicielles), nous optons dans notre approche pour la fusion de ces flots de contrôle pour n'en garder qu'un seul (à partir d'un port de sortie ou d'un port d'entrée de l'entité logicielles où l'interférence a été détectée). Nous avons construit à partir de ces modèles de fusion, un nouveau modèle de fusion qui s'applique sur notre modèle de graphe. Il s'agit de réécrire une partie de ce graphe pour supprimer les points d'indéterminisme ainsi que les accès concurrents. La réécriture d'un graphe g_i en un graphe g_j consiste à effectuer au moins l'une des ces opérations : (1) ajouter des nœuds et/ou des arcs, (2) supprimer des nœuds et/ou des arcs, (3) modifier des étiquettes.

La fusion des flots de contrôle utilise principalement la connaissance de la sémantique des connecteurs. Cette fusion vise à reproduire dans un nouveau flot de contrôle les comportements qui ont été spécifiés par les flots qui ont provoqué le problème d'interférence. Donc cette fusion ne change pas les comportements internes des nœuds mais propose de les interconnecter différemment. Nous appelons

FIGURE 3.8 – Modèle de résolution selon les approches classiques

cette opération une *réécriture structurelle*. La figure 3.8 montre que dans le cas d'une résolution classique, toutes les entités d'adaptation sont conformes à un même modèle. Ce modèle définit les entités logicielles qui composent le système ainsi que l'ensemble de connecteurs de base qui définissent les interactions entre ces entités. Le mécanisme de gestion des interférence est alors conçu en partant de la connaissance de la sémantique de ces connecteurs. Le mécanisme de résolution *Res* détermine le résultat de la fusion de ces modèles en entrée en y incluant la gestion des problèmes apparus lors de leur composition. Le résultat de l'étape de gestion des interférences est conforme à ce même modèle. Cela signifie que l'ensemble des connecteurs utilisés dans la spécification des entités d'adaptation apparaît aussi dans le système finale. Il s'agit d'une transformation homogène [MVG06]. Cette transformation est réalisée en s'appuyant sur le principe de réécriture de graphes qui fera l'objet de la section suivante.

3.3.1.1 Principe de réécriture de graphes

La réécriture de graphes [ENRR88] [RE99] est un mécanisme appliqué pour transformer un graphe d'une manière rigoureuse en s'appuyant sur des concepts mathématiques. Ce mécanisme constitue une base formelle utilisée dans la résolution de diverses problématiques (compilateur, reconnaissance d'objets, etc.). Dans la littérature nous retrouvons deux techniques qui sont les grammaires de graphes et la transformation de graphes qui sont très proches, avec tout de même une différence quant à leur finalité. La technique des grammaires de graphe définit un ensemble de production de grammaire (s'inspirant des grammaires génératives de Chomsky [Cho56]) dont l'intérêt est de produire un ensemble de graphes à partir d'un graphe de départ suite à l'application d'un ensemble des productions. Les nœuds de graphes sont classés en des nœuds terminaux et d'autres non terminaux. Tant que le graphe contient des nœuds non terminaux, les productions seront applicables. Le processus s'arrête alors lorsque le graphe final ne contient plus de non terminaux. La technique de transformation de graphe, quant à elle, s'intéresse à un processus de calcul. Dans ce cas, on parle des règles de transformations qui ont pour but de faire passer un graphe d'un état courant vers un graphe représentant un nouvel état. Tous les nœuds du graphe sont des terminaux. Nous nous intéressons dans la suite à la technique de transformation de graphe qui répond au mieux à notre besoin de réécriture structurelle. Il existe plusieurs types de règles de transformations dont les plus utilisées sont *SPO (Single-PushOut)* et *DPO (Double-PushOut)*.

SPO (Single-PushOut) Ce type de règle [EKL91] exprime le moyen le plus basique pour transformer un graphe g en un autre graphe g'. Une règle est définie par le couple $(L;R)$ et permet de remplacer le sous-graphe L dans g par le sous-graphe R ce qui permettra d'avoir le graphe résultat g'. Le graphe g est le graphe hôte, L est le graphe mère et R est le graphe fille. L'application d'une règle de ce type engendre la suppression de l'occurrence de L dans g et son remplacement par une copie de R. Le

problème posé dans cette approche est l'apparition des arcs suspendus (arcs sans nœud de départ ou sans nœud d'arrivée ou les deux). Dans le cas de règles de transformation SPO, la solution consiste à supprimer ces arcs. De ce faite, l'application d'une transformation SPO peut entraîner l'apparition d'un nœud sans aucun arc vers le reste des nœuds de l'application. Le problème d'arc suspendu a été traité différemment dans l'approche *DPO*.

DPO (Double-PushOut) Une règle de type DPO [EKL91] est un triplet $(L;K;R)$. Les définitions des graphes L et R sont les mêmes que dans le paragraphe précédent. Le sous-graphe K est inclus à la fois dans L et dans R et est utilisé pour expliciter la partie du graphe qui doit être conservée après sa transformation. Dans cette approche il ne suffit pas de trouver une occurrence de L dans g mais il faut satisfaire une autre condition : c'est la condition de suspension. Une règle ne s'applique pas si elle conduit à un problème d'arc suspendu.

La figure 3.9 montre un exemple du problème d'arc suspendu ce qui rend l'application de la règle de type DPO impossible (seulement la règle SOP s'applique dans ce cas). Que se soit SPO ou bien DPO, les deux approches se basent sur la vérification de l'existence du graphe mère dans le graphe hôte : c'est la recherche d'un homomorphisme [Gue06] entre ces deux graphes. Les règles de type DPO nécessitent un double homomorphisme de graphe ce qui rend leur complexité plus importante que celle des règles SPO. Le choix d'un type de règle ou l'autre est déterminé en se référant à la signification de la suppression d'un nœud du graphe et le traitement de ses arcs suspendus.

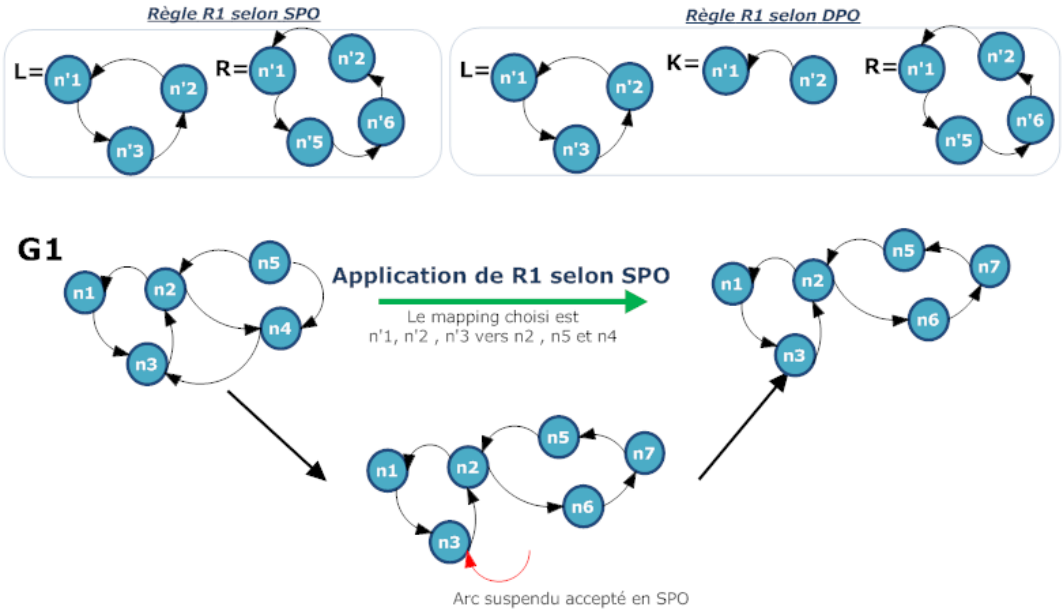


FIGURE 3.9 – Une règle exprimée selon SPO et DPO et étapes de son application

Pour déterminer l'applicabilité d'une règle de transformation, l'approche SPO se base sur l'existence d'une occurrence du sous-graphe L dans g. Dans certains cas, il est nécessaire de pouvoir exprimer des conditions supplémentaires permettant de spécifier des conditions relatives à l'absence d'une occurrence dans le graphe hôte. Ce type de condition est appelé condition d'application négative (Negative Application Condition ou NAC). L'intégration de condition de type *NAC* ajoute donc un nouvel

élément à une règle de réécriture. Une règle *SPO* aura la structure $(L;R;NAC)$ et sera applicable à un graphe g s'il y a une occurrence de L dans g et s'il n'y a pas d'occurrence de NAC dans g. Le sous-graphe NAC n'intervient qu'au niveau de l'applicabilité de la règle et non pas dans l'étape de transformation ce qui permettra de résoudre le problème d'application récursive des règles de transformation (l'application d'une règle produira le sous-graphe L de cette même règle ce qui permettra son applicabilité de manière infinie).

↪ Nous avons vu précédemment que nous avons deux classes de nœuds. Nos règles de transformations manipulent seulement les nœuds de sémantique connue (les connecteurs). Si une règle supprime un nœud alors les arcs entrants ou bien sortants de ce nœud perdent leurs utilité (au niveau de la composition). Cela implique la suppression des arcs suspendus au lieu de désactiver l'application de règles de transformation. Pour toutes ces raisons, nous optons pour l'utilisation des règles de type SPO.

La définition d'une règle de réécriture se base principalement sur la sémantique des connecteurs. Dans la section suivante, nous présentons nos règles d'une manière générique en détaillant la logique de réécriture qui pourra exister entre les connecteurs. Ces règles génériques sont données à titre d'exemple sans se limiter à une sémantique particulière de connecteur (dans le chapitre 4 nous exposons les règles spécifiques pour nos connecteurs).

3.3.1.2 Des règles de réécriture génériques

L'ensemble de règles de transformations que nous proposons s'appuient sur la connaissance de la sémantique des connecteurs. Ces règles agissent sur ces nœuds en modifiant : leurs interconnexions, leurs positions dans le graphe, etc. Dans cette section nous détaillons les opérations de réécriture que nous proposons dans notre approche.

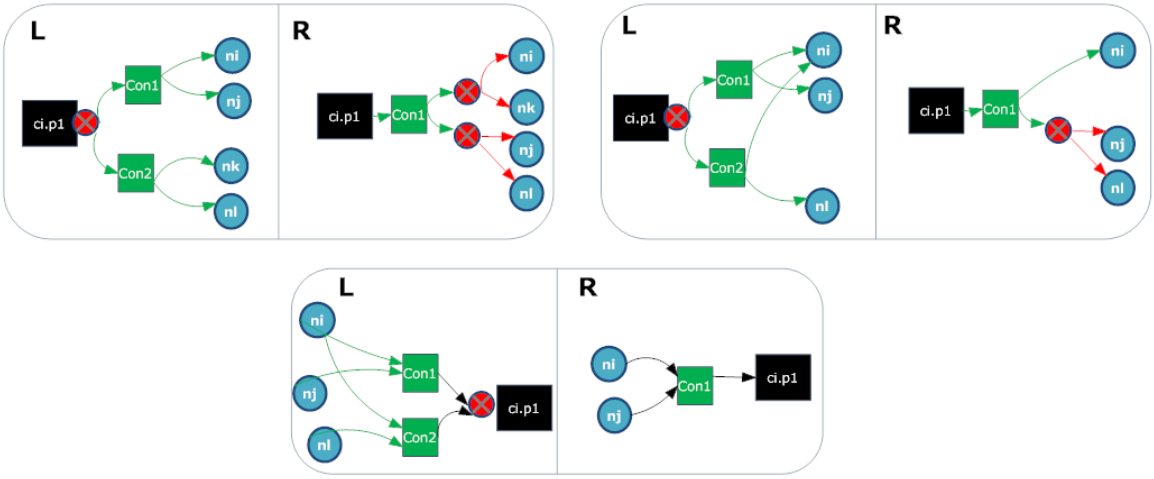


FIGURE 3.10 – Règles de réécriture génériques qui gardent un seul connecteur parmi ceux qui sont en interférence

Garder un seul connecteur Ce type de règle, illustré par la figure 3.10, propose la réécriture de deux connecteurs en un seul. Dans cette figure, les nœuds n_i, n_k,n_j n_l peuvent être des connecteurs

ou bien des ports (*BlackBoxEntity*). Le nœud $\otimes$ est utilisé pour marquer les endroits où il y a un problème à résoudre. Les deux flots de contrôle dans le sous-graphe L vont être réécrits en un seul. La réécriture de ces deux flots garde un seul connecteur avec propagation du nœud $\otimes$ sur les arcs sortant de ce nœud (sous-graphe R). La logique de la combinaison des arcs sortants ou bien des arcs entrants est fortement liée à la sémantique des connecteurs impliqués dans l'opération de réécriture. La règle présentée dans la figure 3.10 à gauche représente une règle de transformation générique pour les connecteurs de type $1-n$. Plusieurs cas particuliers peuvent avoir lieu et se traduisent par d'autres règles de transformation toujours en suivant la même logique de réécriture. Dans le cas où $n_i = n_k$, une autre règle de transformation sera appliquée. Cette règle est donnée par la figure 3.10 à droite. Le nombre de combinaisons possibles dépend aussi de la sémantique de ce connecteur. La figure 3.10 en bas illustre la règle générique pour les connecteurs de type $n-1$.

Modifier la position dans le graphe de l'un de deux connecteurs Le principe de réécriture proposé dans cette règle est d'imbriquer un flot de contrôle dans un autre. Pour ce faire, l'un de deux connecteurs (par exemple *Con*2) va être réécrit pour faire partie de l'ensemble des nœuds successeurs de l'autre connecteur (*Con*1). La règle de transformation correspondant à cette logique est donnée par la figure 3.11.a). Comme dans la première classe de règles de transformation, plusieurs cas de figure peuvent se présenter pour cette classe (par exemple $n_i = n_l$). Cette fois-ci, le choix de la sous branche qui va être réécrit avec le connecteur *Con*2 s'appuie sur la notion de pivot. Le pivot permet de contrôler et guider la réécriture afin qu'elle se fasse à un endroit précis dans le graphe. La partie (b) de la figure 3.11 présente cette règles pour la deuxième classe de connecteurs (type $n-1$).

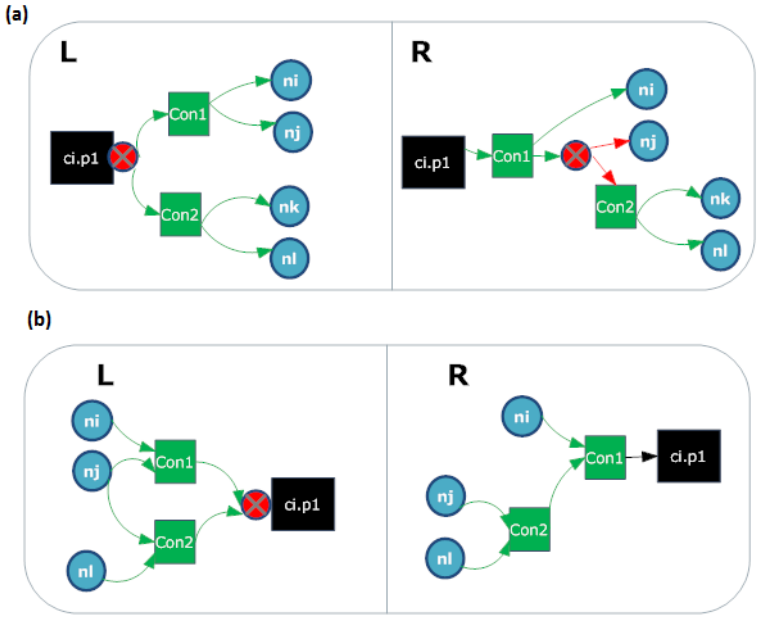


FIGURE 3.11 – Règles de réécriture générique qui modifient l'interconnexion des connecteurs

Dupliquer l'un de deux connecteurs Contrairement aux premières règles qui gardent un connecteur parmi les deux, la règle illustrée par la figure 3.12 duplique l'un de deux connecteurs. Cette opération consiste en la création d'un nouveau connecteur de même type que celui à dupliquer (ici

c'est le connecteur *Con*2). Le nouveau nœud aura des arcs sortants vers les mêmes nœuds que celui du connecteur originaire.

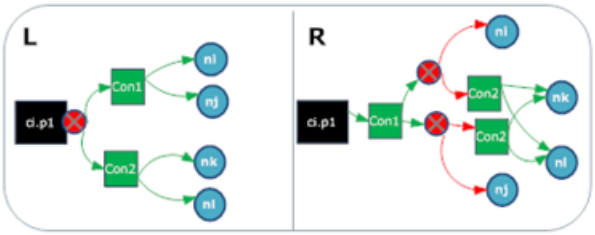


FIGURE 3.12 – Résolution par duplication d'un connecteur

Toutes les règles de transformation présentées dans cette section seront détaillées dans le chapitre suivant en utilisant des exemples concrets de connecteurs ce qui permettra de mieux comprendre la logique de l'ensemble des opérations proposées.

D'autres approches de résolution intègrent dans leur raisonnement des opérateurs de langage. Dans la section suivante, nous présentons cette catégorie d'approches et nous exposons par la suite les limites de ces deux premières catégories de résolution.

3.3.2 Résolution avec l'ajout des opérateurs de langage

Cette approche de résolution est complémentaire à l'approche de résolution classique. Elle utilise aussi le principe de réécriture de graphe. La particularité dans ce type d'approche est la définition des opérateurs de langage qui ont une sémantique et une utilisation spécifique. La figure 3.13 montre que le mécanisme de résolution dans ce cas considère en entrée des entités d'adaptation conformes à un même modèle (comme celui présenté précédemment) avec en plus un ensemble d'opérateurs lié au langage utilisé pour décrire ces entités. La figure montre que ces opérateurs ne font pas partie du modèle (ce sont pas des connecteurs au sens du modèle d'application) et qui sont utilisés seulement pour modifier la logique de résolution des interférences. Ils ne seront jamais présents dans la composition du système final mais ils sont plutôt utilisés en interne par le mécanisme de gestion d'interférence. L'utilité de ces opérateurs est de piloter depuis l'externe (par rapport au mécanisme de résolution) la manière dont la résolution se fait.

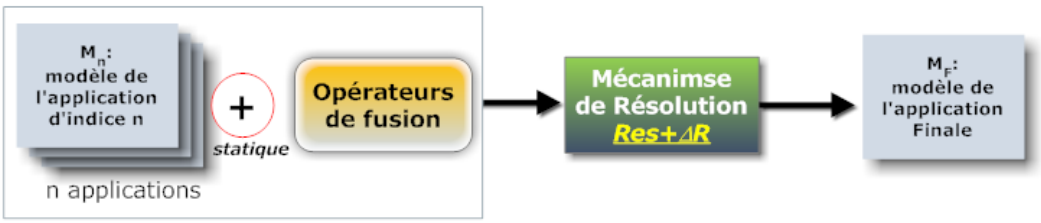


FIGURE 3.13 – Modèle de résolution avec l'ajout des opérateurs de langage

Divers exemples d'opérateurs ont été fournis par les différents langages qui ont été utilisés pour la spécification des compositions. Les langages classiques d'aspect, comme par exemple AspectJ, définissent les opérateurs *Before*, *After* et *Arround*. Ces opérateurs sont utilisés par le tisseur d'aspect pour ordonner les aspects au niveau des points de jonction partagés. Berger dans [Ber01] a défini un

nouvel opérateur qui est *delegate*. *"Cet opérateur spécifie qu'un comportement ne comportant pas le message déclencheur de la règle réactive soit traité comme étant ce message déclencheur"*. Cheung dans [CFW09] a réutilisé le *delegate* et a ajouté l'opérateur *call*. Dans notre approche, nous définissons trois connecteurs *utilitaires* qui seront utilisés seulement pour diriger l'opération de réécriture de graphe :*CALL*, *DELEGATE* et *KILL*. L'opérateur *CALL* est utilisé pour la réécriture d'un lien existant dans la composition de base (c'est dans le cas de la modification de la composition existante). Dans certains cas de spécification de composition, nous avons besoin d'un moyen pour exprimer qu'un lien vers une entité logicielle doit être unique. Cela signifie qu'un port d'une entité logicielle est le seul autorisé à se connecter à un autre port d'une autre entité par exemple. C'est le rôle de l'opérateur *DELEGATE* qui force la suppression de tous les liens qui partagent le nœud source (ou bien destination) avec le *DELEGATE*. Le dernier connecteur utilitaire que nous définissons dans cette classe est *KILL* et est utilisé pour interdire des interactions entre des entités logicielles.

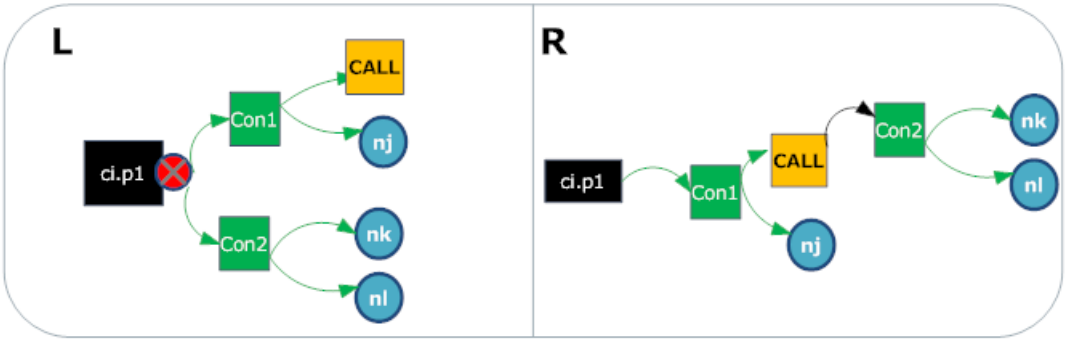


FIGURE 3.14 – Un exemple de règles de réécriture d'un connecteur possédant l'opérateur *CALL* comme successeur

Le mécanisme de gestion des interférences défini dans la section précédente doit être modifié pour inclure ces opérateurs dans sa logique interne de résolution. Cette modification est assurée pendant la phase de conception du mécanisme d'une manière statique (plus de détail dans le chapitre 4 section 4.1). Comme exemple, nous présentons par la figure 3.14 la règles de réécriture dans le cas où un opérateur *CALL* a été identifié. Cette règles est plus prioritaire que celle qui a été spécifié pour la résolution des interférences entre les connecteurs *conn*1 et *conn*2. Le mécanisme de gestion des interférences a été incrémenté par l'ajout des nouvelles règles de réécriture de graphe et la modification des priorité entre toutes les règles existantes.

Les limites de ces deux approches

La spécification des entités d'adaptation qui utilisent des opérateurs de langage fait appel à des hypothèses concernant la composition existante du système. Par exemple l'opérateur *CALL* est utilisé pour réécrire les liens existants. L'utilisation de cet opérateur n'a aucun sens si de telles interactions ne sont pas déjà existantes dans le système. Cela permet d'anticiper au moment de la spécification de la composition, des cas possibles d'interférences qui peuvent ne jamais exister.

Un autre problème dans ce type de résolution est la possibilité d'utiliser ces opérateurs par les différents développeurs sans que chacun ne soit conscient de ce qui a été spécifié par les autres (ε_4). Chacun d'eux introduit, via l'utilisation de ces opérateurs, une logique pour diriger l'opération de la résolution. Par conséquent, des interférences peuvent apparaître à l'issue de l'utilisation de ces opérateurs. Ces problèmes restent non résolus et nécessitent l'intervention des développeurs eux même

(par exemple interférence entre deux *DELEGATE*).

La deuxième approche est une incrémentation de la première. Cette incrémentation est effectuée d'une manière statique c'est à dire qu'il sera nécessaire d'arrêter le mécanisme de gestion des interférences, d'effectuer ensuite les modifications et de redémarrer le mécanisme pour fonctionner de nouveau. Ce mécanisme se base sur un ensemble de connecteurs fixés a priori. Nous avons évoqué que dans notre cadre de travail, les entités logicielles sont multiples (ε_1) et très variables (ε_3). L'apparition de nouvelles entités logicielles pourra provoquer le besoin d'utilisation de nouveaux connecteurs qui implémentent la logique nécessaire pour leur coordination. Avoir un mécanisme de résolution uniquement extensible statiquement freine l'introduction des nouveaux connecteurs de coordination de ces entités logicielles. Un mécanisme de résolution prédéfini à l'avance ne sera pas capable de réagir et résoudre des interférences apparues à l'issue d'utilisation des connecteurs non connus a priori lors de sa conception.

Nous avons donc besoin d'un mécanisme de gestion des interférences qui puisse être étendu automatiquement et dynamiquement. *Est-il possible, sous certaines hypothèses, d'étendre ce mécanisme dynamiquement pour considérer de nouveaux connecteurs ?* Nous proposons dans la suite une méthodologie permettant de définir un tel mécanisme.

3.3.3 Résolution d'interférences par modélisation comportementale

Le mécanisme de résolution que nous avons défini jusqu'à présent suppose l'existence d'un ensemble de connecteurs dont la sémantique est connue a priori. Dans notre contexte de travail, le besoin de définir une nouvelle logique de coordination d'entités peut apparaître et se traduire par l'ajout d'un nouveau type de connecteur. Pour cela la connaissance a priori d'un ensemble figé de connecteurs n'est pas satisfaisant. Dans cette section, nous traitons le cas où il y a eu utilisation d'un connecteur n'appartenant pas à l'ensemble des connecteurs connus. Le mécanisme de résolution devra réagir pendant l'exécution et proposer une solution aux interférences *comme si ce connecteur était connu à l'avance*. Le problème provient ici du fait que le mécanisme de résolution n'a pas une règle de réécriture qui pourra être appliquée à ce nouveau connecteur. Notre objectif est donc **d'étendre le mécanisme de gestion des interférences dynamiquement** sans l'intervention du développeur pour supporter l'apparition des connecteurs non connus.

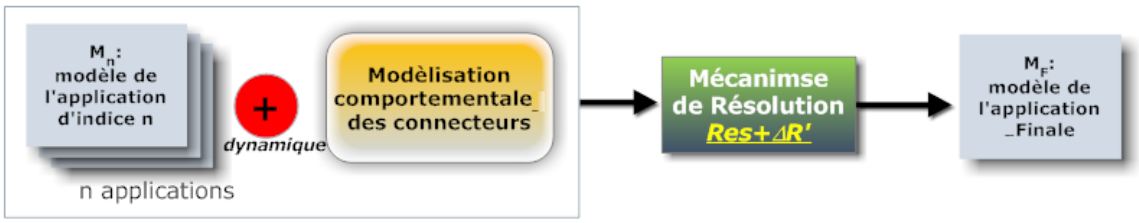


FIGURE 3.15 – Le modèle de résolution avec une modélisation comportementale

Dans l'optique d'atteindre cet objectif, **nous nous basons sur l'hypothèse que ces connecteurs fournissent une description de leur comportement**. Nous utilisons cette description comportementale pour étendre notre mécanisme de résolution qui sera capable par la suite de résoudre les interférences à l'issue de l'utilisation de ces nouveaux connecteurs. *Nous montrons notre démarche en utilisant des connecteurs de type* $n-1$ qui seront ajoutés dynamiquement. Cette classe de connecteurs exprime une logique de déclenchement d'un port de sortie en fonction de l'état des ports d'entrée. La

figure 3.15 montre que notre mécanisme aura en entrée un ensemble d'entités d'adaptation conforme au même modèle ainsi que les modèles comportementaux de chacun des nouveaux connecteurs utilisés par ces entités. Le résultat produit par le mécanisme de résolution sera conforme à ce même modèle. Les nouveaux connecteurs ne seront pas utilisés dans la composition du système final mais réécrits à l'aide de connecteurs de base. Pour commencer, nous définissons le modèle comportementale qui pourra être utilisé par notre mécanisme de gestion des interférences qui sera *extensible dynamiquement*.

3.3.3.1 Présentation de modèles de comportement

Dans cette section nous présentons deux modèles qui sont fortement utilisés pour décrire le comportement de systèmes logiciels.

Réseaux de Petri Les réseaux de Petri (RdP)[Rei85] ont été inventés par Carl Adam Petri en 1960. Il a défini un outil mathématique puissant permettant de : (1) décrire les relations existantes entre des conditions et des événements et (2) modéliser le comportement de systèmes dynamiques. La définition d'un réseau de Petri revient à spécifier les ensembles d'états et des transitions entre ces états. Les états d'un réseau de Petri sont ses marquages et les transitions entre états sont réalisées par le tir de transitions du réseau. Ces transitions indiquent comment et sous quelles conditions le système passe d'un état à un autre.

Automates Les automates [Arn92] sont des objets mathématiques, très utilisés en informatique, qui permettent de modéliser le comportement d'un grand nombre de systèmes. Même si les automates ont été principalement liés à la théorie des langages, il existe de nombreux travaux qui les utilisent pour modéliser des systèmes parallèles (communicants, concurrents). Un automate à états finis est formé d'un ensemble d'états du système, reliés entre eux par des transitions qui sont marquées par des symboles. Ces symboles illustrent les actions du système. Un automate représente alors un système qui évolue au cours du temps et qui réagit par des actions selon son état courant et l'occurrence de certains évènements qui s'appliquent à partir de cet état courant (possibilité de transition vers un autre état).

3.3.3.2 Les éléments du modèle utilisé

Pour pouvoir gérer les interférences apparues à l'issue de l'utilisation des nouveaux connecteurs, nous avons besoin d'une description de leur modélisation comportementale. Un connecteur reçoit des évènements de ses port d'entrées et produit un résultat par l'envoi d'un événement sur ses ports de sortie. Un connecteur dispose d'un état interne qui évolue en fonction des évènements présentées sur ses ports d'entrée. Une séquence d'événements appliquée à l'entrée d'un connecteur conduit celui-ci à évoluer au sein d'un espace fini d'états. Ainsi, à tout moment, l'état interne du connecteur reflète un historique des événements qui se sont présentés précédemment. C'est ce mécanisme qui permet à un connecteur de produire des états de sortie différents en réponse à un même événement dont les occurrences se situent dans des moments différents. Dans notre approche, nous utilisons le modèle d'automates à états finis (AEF) qui est adapté à notre problématique. Nous avons déjà évoqué dans l'étude de l'état de l'art l'importance du modèle d'automate qui a été utilisé dans [MSA06]. Le choix de ce modèle est justifié aussi par [Arn92]. Ce modèle permet d'exprimer les relations d'ordre entre les opérations qui consistent à définir à quel moment ils peuvent s'effectuer. Ce moment correspond à un état dans l'automate. Un cas particulier d'automate est celui des *automates communicants* qui est

très utilisé dans les systèmes parallèles et/ou distribués. Les automates communiquent par échange de messages. Des étiquettes sont utilisées sur les transitions pour marquer l'envoi et la réception des messages. Le message peut être le nom d'un signal ; nous parlons dans ce cas de message de contrôle. Il peut spécifier aussi des paramètres, on parle alors de message de données.

Les choix d'architecture Le comportement d'un AEF peut être défini par un graphe d'états. La représentation de ce graphe dépend de l'architecture du modèle choisi : *Machine de Mealy* ou *Machine de Moore*. À titre de comparaison, la figure 3.16 présente l'architecture de ces deux modèles. Dans une machine de Mealy (figure 3.16.a) la sortie dépend de l'état courant et des entrées (la fonction F dans la figure). Une machine de Moore (partie b de la figure) est un système de transition dont les sorties ne dépendent que de l'état. Cette figure montre bien qu'une machine de Moore n'est qu'une restriction de la machine de Mealy puisque la fonction de sortie F ne reçoit plus le vecteur d'entrée E. De ce faite, les valeurs de sortie ne sont plus directement liées aux entrées et elles ne dépendent que de l'état interne de la machine. Au plan comportemental, cette restriction implique qu'il n'est plus possible, comme dans le cas de l'automate de Mealy, de conditionner directement les valeurs de sortie à celles des entrées de l'automate. Pour une machine de Mealy, la fonction G sert à calculer l'état futur en se basant sur E et l'état courant. L'état futur est enregistré dans un registre (flip flop). Pour tenir compte de l'action immédiate des entrées sur les sorties, dans une machine de Mealy, on complète parfois le diagramme de transitions de la machine en faisant figurer, en plus de la condition de transition, la valeur associée des sorties du type Mealy. Malgré cette différence entre les deux modèles, il est toujours possible de transformer un automate d'un modèle à un autre (tout problème peut être résolu par une machine de type Mealy ou Moore, indifféremment). Normalement, pour un même problème, une machine de Moore demande plus d'états que la machine de Mealy équivalente. La machine de Moore est donc potentiellement moins rapide que la machine de Mealy. *Nous utilisons alors une représentation des automates selon le modèle Mealy*. Le graphe d'états selon ce modèle spécifie les événements régissant les transitions entre les différents états internes de l'automate. Chaque transition est matérialisée par un arc étiqueté par deux types d'attributs binaires : d'une part, les valeurs des entrées de l'automate associées à la transition, et d'autre part, les valeurs des sorties associées à cette même transition.

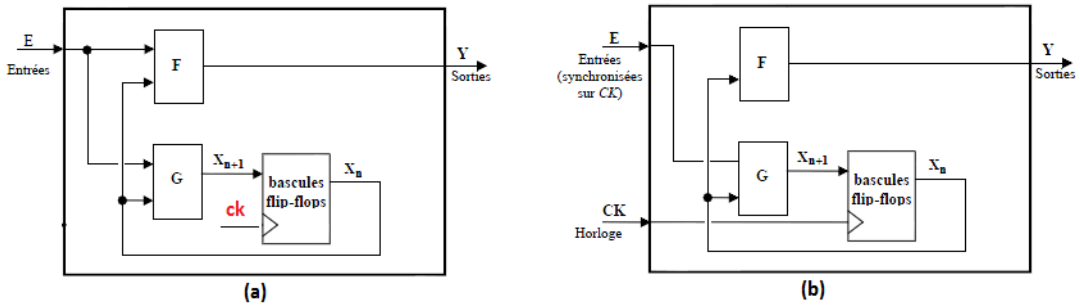


FIGURE 3.16 – (a) Architecture de machine de Mealy (b) Architecture d'une machine de Moore

La machine de Mealy évolue généralement selon le modèle asynchrone. Cela présente un risque d'ambiguïté dans le résultat de sortie du connecteur puisque les entrées et la sortie de ce dernier ne suivent pas la même dynamique. Ce problème est résolu par l'utilisation du modèle synchrone qui permet de maîtriser plus aisément le comportement et l'exploitation du connecteur. Le principe de la synchronisation consiste à utiliser un signal, en général périodique, appelé *horloge*, pour rythmer

l'évolution du système (ceci justifie l'ajout du signale *ck* dans la figure 3.16.a). Une machine de Mealy peut aussi présenter un comportement synchrone si on ajoute une entrée horloge. Le rôle d'une horloge dans une réalisation synchrone, est de supprimer toute possibilité d'aléas dans l'évolution de l'état. Idéalement, entre deux transitions actives de l'horloge le système est figé, en position mémoire, son état ne peut pas changer. Dans ce modèle, chaque réaction est atomique et définit un instant. Une réaction n'a pas de durée puisque sa durée réelle est retardée pour le prochain cycle d'horloge ou l'instant suivant du système. Autrement dit, les sorties sont considérées comme simultanées aux entrées qui les provoquent.

Le comportement d'un connecteur est défini par la machine de Mealy suivante : $Con = \prec Q, q_0, F, I, O, \mathcal{T}, \mathcal{L} \succ$, où Q est un ensemble fini d'états ; q_0 l'état initial, F est l'ensemble des états finaux ($F \subset Q$), I est un ensemble fini d'événements d'entrée, O est l'ensemble fini d'événements de sortie. $\mathcal{T} \subseteq Q \times Q$ est une relation de transition et $\mathcal{L}$ est la fonction d'étiquetage des transitions. Les transitions ont la forme suivante : $q \xrightarrow{f/o_1,\ldots,O_n} q_1$, où f est une fonction booléenne qui utilise uniquement les événements d'entrée et o_i sont des événements de sortie. $f ::= x|f \vee f|f \wedge f|\bar{f}x \in I$.

Chaque machine de Mealy peut être représentée par un graphe dirigé : les sommets du graphe sont les états de la machine, et on dessine une flèche de l'état q à l'état p étiquetée par e/s si et seulement si il existe une transition $q \xrightarrow{e/s} p$ dans $\mathcal{T}$. Le comportement de nos connecteurs doit être déterministe ce qui signifie que pour tout couple $< q_i, q_j >$ le choix de la transition est unique (il est indéterministe s'il existe un état qui a au moins deux transitions possibles pour un même événement d'entrée qui mènent vers deux états différents). Si $q \xrightarrow{i_1/o_1} q_1$ et $q \xrightarrow{i_1/o_2} q_2$ alors, $q_1 = q_2$ et $o_1 = o_2$.

Ce modèle comportementale de connecteur va servir pour l'extension de notre mécanisme de gestion des interférences.

3.3.3.3 Méthodologie pour la génération des règles de réécriture de graphe

Nous avons vu que le mécanisme de gestion des interférences se base sur un ensemble de règles de réécriture prédéfinies pour résoudre les problèmes identifiés. Ces règles expriment comment réécrire les connecteurs deux à deux en s'appuyant sur la connaissance de leurs sémantiques. L'utilisation d'un connecteur en dehors de cette liste préétablie de connecteurs rend notre mécanisme de gestion des interférences incapable de résoudre les interférences pour tout nouveau connecteur. L'idée ici est de définir une méthodologie permettant de générer les règles de réécriture d'une manière automatique pendant l'exécution du mécanisme de gestion des interférences (*C'est ce que nous appelons une extension dynamique*). La figure 3.17 illustre le processus pour atteindre l'objectif de l'extension.

Pour ce faire, nous composons l'automate du nouveau connecteur avec chacun des connecteurs de base. L'automate résultant de cette opération de composition sera exploité pour générer la règle de réécriture du nouveau connecteur avec chacun des connecteurs de base. Dans cette section nous décrivons, tout d'abord, l'opération de composition des automates et ensuite, nous présentons les étapes pour la génération d'une règle de réécriture à partir d'un automate.

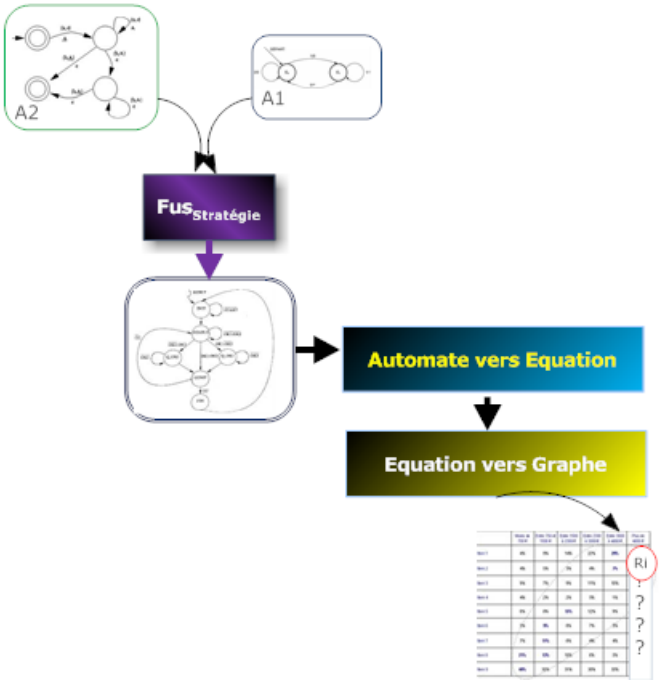


FIGURE 3.17 – Le processus pour la génération des règles de réécriture de graphe

Composition des automates

La première étape dans notre approche est la composition de l'automate du nouveau connecteur avec l'automate d'un connecteur que nous connaissons déjà. L'état courant de l'automate composé est une combinaison des états courants des automates le constituant. Étant donné deux logiques de déclenchement d'automate de Mealy de chacun des connecteurs, $\mathcal{M}_1$ et $\mathcal{M}_2$, comment peut-on fusionner leurs comportements dans un seul $\mathcal{M}_3$? Il existe plusieurs opérations sur les automates : différence, intersection, masquage, union, etc. L'opération de fusion que nous proposons se base sur le produit synchrone des automates.

Définition 7 : Composition de deux automates
Soit $C1 =< Q_1, q_{01}, F_1, I1 = \{i_{11}, i_{12}\}, out_1, \mathcal{T}_1, \mathcal{L}_1 >$ *et*
$C2 =< Q_2, q_{02}, F_2, I2 = \{i_{21}, i_{22}\}, Out_2, \mathcal{T}_2, \mathcal{L}_2 >$ *deux machine de Mealy.*
La fusion de C1 et C2 produit une machine de Mealy $C3 = C1 \otimes C2 =< Q_1 \times Q_2, q_{01} \times q_{02}, I3 = \{i_{11}, i_{12}, i_{21}, i_{22}\}, Out_3, Sync, \mathcal{L}_1 \cup \mathcal{L}_2 >$.

Le produit synchrone envisage toutes les combinaisons possibles des états des automates $\mathcal{M}_1$ et $\mathcal{M}_2$ sous une contrainte d'ordonnancement $\mathcal{S}ync$ qui restreint les transitions autorisées dans le produit cartésien. L'évènement de sortie de l'automate résultant est une fonction des évènements de sortie de chacun de deux automates : $Out_3 = Fus_{stratgie}(out_1, out_2)$. La fonction $Fus_{stratgie}$ représente la politique qui sera appliquée pour l'appel des ports. Elle peut être par exemple *restrictive* ou *permissive*. Une fusion de deux connecteurs C_i et C_j est dite restrictive si elle cherche à satisfaire la logique de déclenchement à la fois de ces deux connecteurs. Il s'agit d'une composition de type AND ; $Out_3 = [out_1, out_2] \bullet AND$. Cependant une politique permissive utilise une composition parallèle (OR) ce qui se traduit par la satisfaction de l'une des deux logiques de déclenchement $Out_3 = [out_1, out_2] \bullet OR$.

Composition OR (Produit sans synchronisation) Le cas le plus simple à traiter est celui d'automates qui modélisent des sous-connecteurs qui n'interagissent pas entre eux. L'automate global est alors le produit cartésien des automates de ses sous-connecteurs ; un état global est en fait un vecteur des différents états des sous-systèmes (états locaux). Dans ce type de composition, la satisfaction de l'une de deux logiques de déclenchement des sous connecteurs (c'est-à-dire l'émission d'un output de l'un de deux automates) implique l'appel du port du composant en question.

Composition AND (Produit avec synchronisation simple) Le cas que l'on rencontre le plus souvent est plutôt le cas où les sous connecteurs doivent se synchroniser. La composition AND (appelée en programmation concurrente une barrière) de deux automates implique le besoin de la satisfaction de deux logiques de déclenchement. Cela signifie l'émission d'un événement de sortie de chacun de deux automates à composer.

Composition avec synchronisation par des contraintes La définition d'un point de synchronisation, qui gère le moment où une action peut être exécutée, modifie le comportement des connecteurs dont l'exécution doit respecter les contraintes. À titre d'exemple, une contrainte peut imposer un ordre entre les évènements d'entrée de deux connecteurs. Cela se traduit au niveau du produit cartésien, par la suppression des transitions qui ne respectent pas cette contrainte d'ordre.

Le mécanisme de gestion d'interférence doit implémenter une des stratégies de composition. L'utilisation de plusieurs stratégies dans un cycle de tissage [Fer11] risque d'influencer la cohérence du résultat final. L'automate obtenu par la phase de fusion sera exploité par la suite pour déterminer la règle de réécriture correspondant à la fusion de connecteurs en question.

Étapes pour la génération des règles de réécriture

Dans notre démarche nous nous inspirons de la méthode d'*Huffman* utilisée dans la conception des circuits électroniques et qui permet de guider le concepteur à partir du cahier des charges d'un automate jusqu'à l'établissement des équations de la machine de Mealy. Cette démarche permettra d'exprimer le comportement de l'automate en utilisant les opérateurs logique de base. En suivant la même logique, l'automate résultant de l'étape de fusion sera utilisé pour trouver l'équation qui met en relation les évènements d'entrée et les évènements de sortie en utilisant nos connecteurs de base. Le détail de génération des règles de réécriture pour les connecteurs de base est décrit dans section B.1 de l'Annexe B .

De l'automate aux équations L'automate est un graphe dont les nœuds sont les états et les arcs sont les transitions (évolution dynamique du système). L'état actuel à un instant t est déterminé selon l'équation suivante : $Etat\_actuel(t) = f(Etat\_actuel(t-1), Entres(t-1))$; avec f la fonction qui calcule l'état futur qui dépend de l'état présent et des entrées. L'état actuel à un instant t a été déjà enregistré comme étant l'état futur en respectant la relation suivante : $Etat\_actuel(t) = Etat\_futur(t-1)$. Nous rappelons que notre objectif est de trouver à partir de l'automate l'équation régissant le comportement de cet automate en utilisant nos connecteurs de base. À ces connecteurs s'ajoutent alors des connecteurs de mémorisation d'état futur. Nous nous inspirons des langages synchrones, et nous appelons ce connecteur *Next* (en circuit logique ce sont les bascules). Le nombre de connecteurs *Next* nécessaire constitue la taille du registre d'état. Si n est le nombre de connecteur *Next* et N est le nombre d'états de l'automate, alors ces deux nombres sont reliés par la relation $N = 2^n$

Une fois nous avons déterminé le nombre de connecteurs *Next*, nous avons besoin d'attribuer des codes à ces connecteurs ainsi qu'aux évènements d'entrée.

Codage des états Pour passer d'un diagramme de transitions aux équations de sortie et du *Next*, nous pouvons toujours recourir à une table de vérité qui récapitule toutes les transitions possibles. La première étape à faire est le codage des états de l'automate. À titre d'exemple, pour un automate de 4 états $A, B, C et D$ nous utilisons les codes suivants :

Etat	*code*
A	00
B	01
C	10
D	11

La table de vérité La table de vérité met en relation l'état présent, les entrées, l'état futur ainsi que les sorties. Un exemple de table de vérité pour un automate à 4 états, deux entrées et une sortie est illustré dans la figure 3.18.

Etat présent	Entrées	Etat futur	Sorties
Q1 Q2	E1 E2	Next1 Next2	S1

FIGURE 3.18 – Table de vérité obtenue à partir de l'automate

La table de vérifié est remplie en parcourant les états de l'automate. Pour chaque état présent (sous forme Q_1Q_2, avec $Q_1 = \{0,1\}$ et $Q_2 = \{0,1\}$), étant donnée les entrées E_1 et E_2, nous obtenons l'unique état futur (sous forme Nex_1Next_2, avec $Next_1 = \{0,1\}$ et $Next_2 = \{0,1\}$) ainsi la sortie S_1.

Réalisation de la machine à l'aide de composants élémentaires La reproduction du comportement de l'automate de Mealy consiste en la synthèse des fonctions G et F (figure 3.16) pour déterminer la sortie et l'état futur à sauvegarder par les registres (Next). Les équations relatives aux fonctions G et F sont obtenues suite à une étape de simplification en utilisant les diagrammes de Karnaugh [Kar53]. C'est un tableau de 2^n cases, n étant le nombre d'entrées de l'automate. La méthode consiste à réaliser des groupements de cases adjacentes contenant des 1 ou des 0. Dans notre approche, un groupement de 1 permet d'obtenir l'équation de la sortie $S1$ et des connecteurs *Next* (les registres). Cette méthode utilise le code Gray ou binaire réfléchi, qui a comme propriété principale de ne faire varier qu'un seul bit entre deux entrées successives. L'équation est obtenue directement en faisant ces groupements.

De l'équation vers la règles de réécriture Les équations simplifiées obtenues dans l'étape précédente, expriment l'évènement de sortie en fonction des évènements d'entrée en utilisant les connecteurs élémentaires. Ces équations seront réécrites selon notre formalisme de graphe. La règle de réécriture obtenue est enregistrée dans la base de règles de notre mécanisme de gestion des interférences et sera réutilisée ultérieurement (sans refaire la même démarche).

3.4 Conclusion

La résolution des interférences que nous proposons s'appuie sur une représentation abstraite des compositions à l'aide des graphes et d'un ensemble de règles de réécriture. Ce chapitre a détaillé diverses approches pour la définition d'un mécanisme de gestion des interférences. Les approches classiques ainsi que celles utilisant des opérateurs de langage présentent certaines limitations. Principalement, les règles de réécriture utilisées pour la résolution des interférences sont définies d'une manière statique et est axée sur un nombre fixe de connecteurs déjà connus et possèdent une sémantique implicite au nom de l'opérateur. Il n'est pas possible de résoudre les interférences en dehors de la liste de connecteurs prévus. Notre objectif était de remédier à ce problème et de permettre une extension dynamique du mécanisme de résolution des interférences dans le cas d'ajout d'un nouveau connecteur. Pour cela nous avons défini une méthodologie permettant de générer d'une manière automatique et dynamique les règles de réécriture nécessaires à la résolution. Pour ce faire, tout nouveau connecteur doit fournir la description explicite de son modèle comportementale. Nous avons utilisé le modèle d'automate. Dans le cas où il y a eu une interférence à l'issue de l'utilisation de ce nouveau connecteur, et si aucune règle de réécriture n'a pu être trouvée, l'automate du nouveau connecteur sera composée avec celui du connecteur en question. Plusieurs opérations de composition ont été définies dont chacune présente une stratégie donnée (par exemple composition parallèle, synchronisé, etc.). L'automate résultant de cette opération de composition sera traduit en un graphe en utilisant les connecteurs de base. De cette manière, nous construisons une nouvelle règle de réécriture pour ce connecteur. La règle obtenue sera ajoutée à la base de règles du mécanisme de gestion des interférences et sera réutilisée pour la résolution des cas similaires.

Le chapitre suivant décrit notre mécanisme de gestion des interférences selon deux niveaux. Le premier niveau détaille le processus de la conception statique pour les connecteurs de base. Le deuxième niveau illustre l'utilisation de ce mécanisme pour la gestion des interférences à l'exécution. Il décrit également le processus d'extension automatique.

CHAPITRE 4

Un modèle de composition des applications à deux niveaux

Sommaire

Ce chapitre présente l'architecture de notre mécanisme de composition qui est construit à partir des contraintes imposées par l'IAm et des constats que nous avons tirés de l'analyse de l'état de l'art. Notre mécanisme de composition se décompose en deux niveaux : *statique* et *dynamique*. La figure 4.1 montre les étapes qui constituent chacun de ces deux niveaux. Dans un premier temps, nous détaillons l'ensemble des opérations qui constituent la phase de conception statique du mécanisme. Une fois validé statiquement, ce mécanisme pourra être utilisé pendant l'exécution pour composer les diverses entités d'adaptation et résoudre par la suite les problèmes d'interférence apparus. Nous verrons ensuite que notre mécanisme ne se limite pas à la logique de résolution qui a été prédéfinie, mais qu'il est capable aussi d'évoluer dynamiquement pour considérer d'autres cas d'interférences qui n'ont pas été prévu au départ. De cette manière, nous obtenons un mécanisme de composition qui supporte à la fois l'extension de l'ensemble des entités d'adaptation ainsi que l'extension de leur expressivité. L'extension de l'expressivité des entités d'adaptation, c'est à dire ajouter de nouveaux connecteurs, est suivi par une extension du mécanisme de gestion des interférences. Tout cela est effectué dynamiquement en garantissant un ensemble de propriétés.

4.1 La conception statique du mécanisme de composition

L'indépendance de la spécification des entités d'adaptation est à l'origine du problème d'interférences. Les approches qui réalisent la composition à la conception des applications comme Adore [Mos10] font appel au *designer* (un développeur) pour qu'il intègre manuellement sa solution aux problèmes identifiés. Le designer joue alors deux rôles : il spécifie les compositions et il résout ensuite les interférences. Nous avons évoqué précédemment le besoin d'une résolution dynamique et

automatique des interférences. Bien que la gestion des interférences soit réalisée à l'exécution de l'application, la définition de la manière dont la résolution est effectuée est réalisée au moment de la conception du mécanisme de composition et non pas au moment de la conception de l'application. Contrairement aux approches qui confient la spécification des compositions et la résolution des interférences à un même acteur qui est le designer, nous introduisons dans notre démarche un nouvel acteur qui est le **Super-Designer**. Ce dernier prend la responsabilité de la conception *statique* du mécanisme de composition qui va intervenir pendant l'exécution de l'application. Le Super-Designer spécifie alors tous les éléments nécessaires pour la description des entités d'adaptation (c'est lui qui spécifie le DSL, section 4.1.1) ainsi que la logique à mettre en œuvre dans le cas où des interférences ont été détectées (sections 4.1.3 et 4.1.2). L'objectif de cette section est de détailler le processus de conception du mécanisme de composition sous forme de tâches qui doivent être accomplies par cet acteur. Toutes ces étapes se basent sur le formalisme de graphe. Nous utilisons dans la suite notre modèle de graphe pour la description de ce processus.

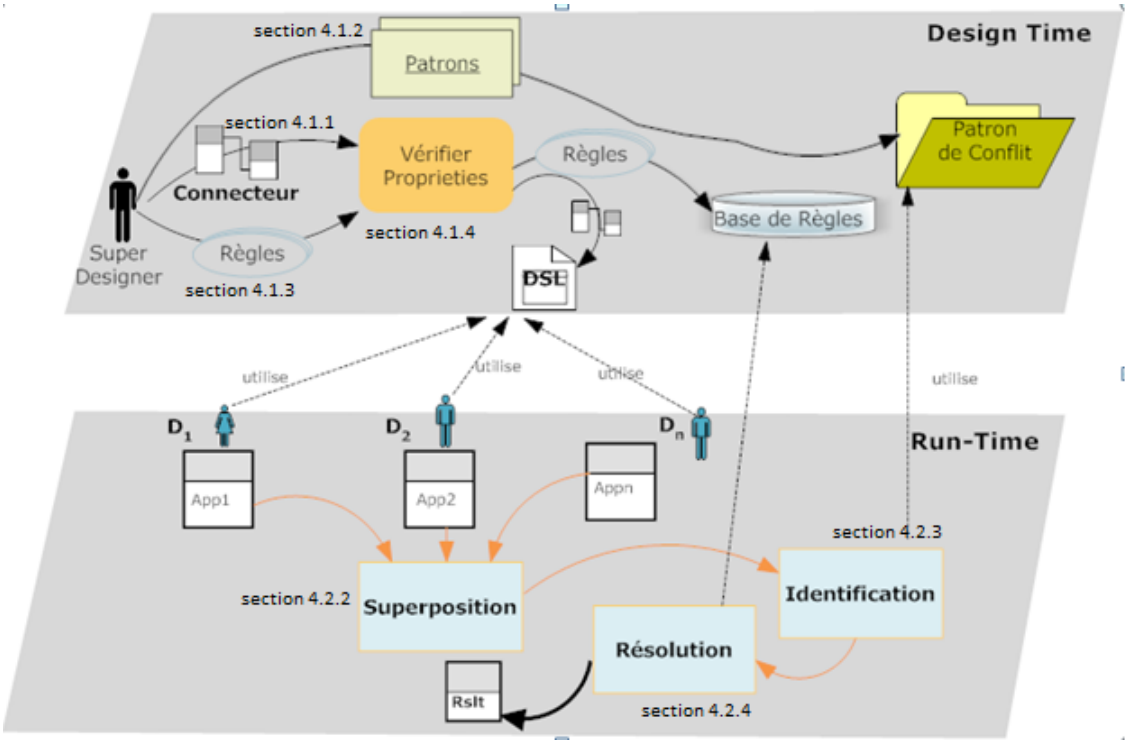


FIGURE 4.1 – Les deux niveaux du mécanisme de composition

4.1.1 Tâche 1 : La définition de la sémantique des connecteurs primitifs

Pour rappel, nous avons vu que notre approche de gestion des interférences se base sur la connaissance de la sémantique de nœuds particuliers dans le graphe de l'application qui sont les *connecteurs*. Pendant la phase de la conception du mécanisme de composition, le Super-Designer définit un ensemble de connecteurs de base qui seront utilisés dans la spécification des entités d'adaptations. L'expressivité d'une composition dépend alors de l'ensemble des connecteurs fournis. La définition d'un connecteur consiste à définir ses ports d'entrée et sortie ainsi que la signification des liaisons qu'il pourra avoir. Nous avons formalisé le comportement de nos connecteurs en utilisant les automates ce

qui nous a permis de valider le résultat de leur composition (plus de détails section 4.1.4).

Dans notre travail nous considérons un ensemble de connecteurs qui dérivent des langages de spécification des compositions (tel que BPEL, ISL4WComp, etc.). À chacune des classes de connecteurs présentés dans la section 3.1.1, correspond un ensemble de connecteurs de base.

Les connecteurs $1-n$**.** Le tableau 4.1 présente les connecteurs appartenant à cette classe [FBALT$^+$12a]. Un connecteur de type $1-n$ définit un point à partir duquel un seul flot de contrôle peut être divisé en plusieurs flots. Ces flots vont être exécutés en parallèle pour le connecteur *PAR* et séquentiellement pour le connecteur *SEQ*. Les connecteurs de cette classe transmettent les données reçues en entrée vers les nœuds successeurs. Un cas particulier de connecteur de cette classe est le connecteur conditionnel *IF*. L'évaluation d'une condition permettra de déclencher un unique chemin de flot de contrôle (la branche *Then* ou bien celle de *Else*).

TABLE 4.1 – Connecteurs de base $1-n$.

Le type CTy	**Notation formelle**	**Description**
PAR	$PAR\bullet[v_i, v_j]$	Définir deux actions en parallèle v_i and v_j
SEQ	$SEQ\bullet[v_i, v_j]$	Définir un ordre entre deux actions v_i ensuite v_j
IF	$IF\bullet[v_c, v_i, v_j]$	Exécution conditionnelle . choisir un chemin entre v_i and v_j

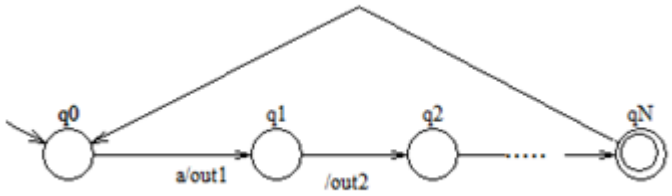


FIGURE 4.2 – Automate du connecteur SEQ

Nous utilisons le modèle d'automate pour définir le comportement de nos connecteurs. Comme exemple, nous montrons dans la figure 4.2 l'automate du connecteur *SEQ*. À la réception d'un événement d'activation de son port d'entrée, les événements de sortie seront transmis les uns après les autres via les ports de sortie de ce connecteur.

Les connecteurs $n-1$**.** Les connecteurs de cette classe utilisent l'une de deux stratégies abordées précédemment (bloquante, non bloquante). Comme exemple de connecteur de type bloquant nous définissons le connecteur *AND*. L'événement de sortie de ce connecteur ne se produit que quand les deux événements d'entrée ont été bien reçus. La figure 4.3 illustre le modèle de comportement de ce connecteur. La deuxième stratégie est non bloquante et représentée par le connecteur *OR*. Cet automate présenté par la figure 4.4 montre bien que l'activation de l'une de deux entrées entraîne l'envoi de l'événement de sortie.

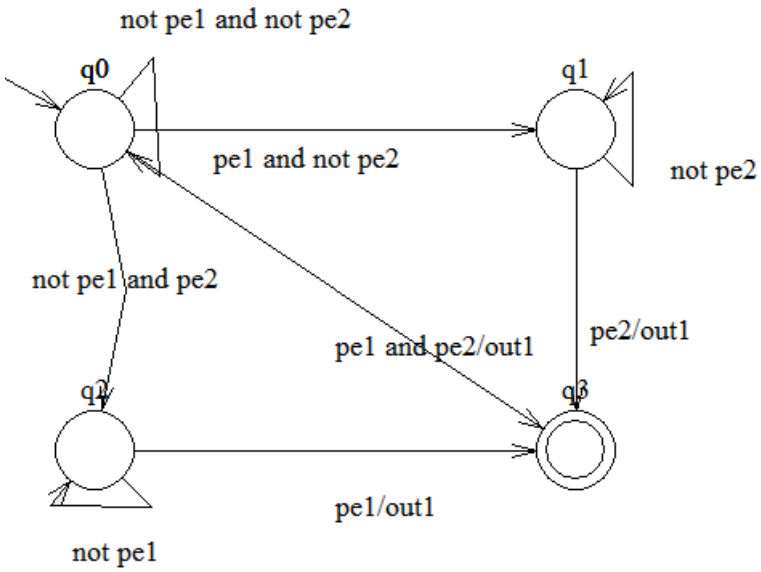


FIGURE 4.3 – Automate du connecteurs *AND*

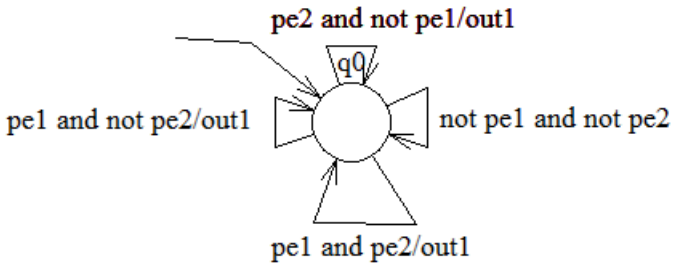


FIGURE 4.4 – Automate du connecteurs *OR*

Les connecteurs utilitaires $1-1$**.** Dans cette classe nous définissons trois connecteurs utilitaires qui sont *CALL*, $DELEGATE$ et *KILL*. Pour rappel, le connecteur *CALL* est utilisé pour la réécriture d'un lien existant dans la composition de base (c'est dans le cas de la modification de la composition existante). Le rôle du connecteur $DELEGATE$ force la suppression de tous les liens qui partagent le nœud source (ou bien destination) avec le $DELEGATE$. Le dernier connecteur utilitaire que nous définissons dans cette classe est *KILL* et est utilisé pour interdire des interactions entre des entités logicielles. Ces connecteurs sont dit utilitaires car ne seront pas présents dans le système finale (section 3.3.2)

↪ **À retenir.** *Les connecteurs qui sont définis au moment de la conception du mécanisme de composition seront utilisés par la suite par tous les designers pour la spécification des entités d'adaptation. Ils forment l'ensemble des connecteurs de base. La définition statique du mécanisme de gestion des interférence se base sur cet ensemble de connecteurs.*

4.1.2 Tâche2 : La spécification des motifs d'interférence

Les problèmes d'interférence qui sont traités pendant la phase de composition se définissent en utilisant le même modèle que celui utilisé pour la définition du système (un graphe). Les types de problèmes que nous traitons ont été présentés précédemment dans la section 3.2. La figure 4.5 illustre les deux patrons génériques que nous traitons dans notre approche. Une interférence est identifiée dès qu'il y a deux arcs sortants ou deux arcs entrants pour un nœud appartenant à l'ensemble B. Les nœuds n_j, n_p, n_l et n_k sont utilisés pour désigner des nœuds de sémantique connue ou les ports d'entités logicielles. Le Super-Designer spécifie pendant la phase de conception du mécanisme de composition les sous-graphes qui représentent des problèmes. L'ajout d'un nouveau patron de conflit est possible pendant cette phase et va engendrer la modification de l'ensemble des opérations qui succèdent cette étape (tâche 3 et tâche 4).

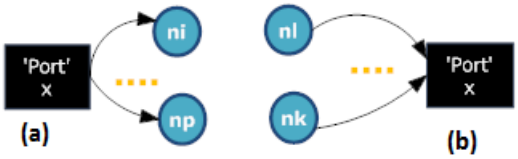


FIGURE 4.5 – Les deux graphes correspondant aux deux motifs d'interférences

Le nombre d'arc sortant (respectivement entrant) d'un nœud appartenant à P_s (respectivement appartenant à P_e) détermine l'ordre d'interférence pour ce nœud. Pour un nœud n_i appartenant à P_s et dont son degré $deg^+(n_i) = p$, l'interférence est d'ordre $p-1$.

4.1.3 Tâche 3 : Spécification des règles de résolution

C'est l'étape la plus importante dans la conception du mécanisme de composition. Une fois que les connecteurs de base ont été validés et que les motifs d'interférences ont été fournis, le Super-Designer décrit à travers un ensemble de règles de réécriture de graphe la logique qu'il va attribuer à son mécanisme de composition. Ces règles décrivent explicitement comment résoudre les interférences en exploitant la connaissance de la sémantique des connecteurs. L'ensemble de règles de transformation que nous proposons dans cette section dérivent des travaux de Cheung [CFW09]. Nos règles de transformation sont binaires c'est-à-dire qu'elles explicitent comment résoudre le problème entre deux nœuds (connecteurs). Elles permettent de résoudre les interférences d'ordre 1 (entre deux nœuds). Dans le cas d'une interférence d'ordre p, au moins $p-1$ règles de transformation seront appliquées. Nous détaillons trois exemples de règles de transformations : une règle pour deux connecteurs de type *1-n*, une autre règle pour deux connecteurs de type *n-1* et une troisième règle pour un connecteur *1-n* et autre *n-1*. L'ensemble des règles utilisées dans notre approche est détaillé dans l'annexe A.

4.1.3.1 Règle $R1$: Deux SEQ

Cette règle de résolution est appliquée dans le cas d'une interférence entre deux connecteurs de Séquence *SEQ*. Chaque connecteur *SEQ* définit un ordre d'exécution entre ces évènements de sortie. La réécriture de deux connecteurs *SEQ* devra conserver les relations d'ordre établies au sein de chacune de deux séquences séparément. Il existe plusieurs configurations possibles pour deux

connecteurs SEQ. Dans le cas ou ces deux connecteurs partagent un nœud, ce dernier pourra avoir un ordre différent dans chacun de deux SEQ. Une configuration possible est $SEQ\bullet[a,b]$ et $SEQ\bullet[a,c]$. Dans cet exemple les deux connecteurs partagent le nœud a qui a le même ordre d'exécution dans les deux connecteurs. Une autre configuration possible est $SEQ\bullet[a,b]$ et $SEQ\bullet[c,a]$. Le résultat attendu par la réécriture de ces deux configurations n'est pas le même. Dans la figure 4.6, nous représentons la réécriture de la deuxième configuration. La résultat est donné par le sous-graphe R de la figure 4.6 : tout d'abord n_i ensuite n_k et finalement n_l. La spécification formelle de cette règle de réécriture selon le formalise défini (section 3.1) est comme suit :
Cette règle de résolution est appliquée dans le cas d'une interférence entre deux connecteurs de Séquence SEQ. Chaque connecteur SEQ définit un ordre d'exécution entre ces évènements de sortie. La réécriture de deux connecteurs SEQ devra conserver les relations d'ordre établies au sein de chacune de deux séquences séparément. Il existe plusieurs configurations possibles pour deux connecteurs SEQ. Dans le cas ou ces deux connecteurs partagent un nœud, ce dernier pourra avoir un ordre différent dans chacun de deux SEQ. Une configuration possible est $SEQ\bullet[a,b]$ et $SEQ\bullet[a,c]$. Dans cet exemple les deux connecteurs partagent le nœud a qui a le même ordre d'exécution dans les deux connecteurs. Une autre configuration possible est $SEQ\bullet[a,b]$ et $SEQ\bullet[c,a]$. Le résultat attendu par la réécriture de ces deux configurations n'est pas le même. Dans la figure 4.6, nous représentons la réécriture de la deuxième configuration. La résultat est donné par le sous-graphe R de la figure 4.6 : tout d'abord n_i ensuite n_k et finalement n_l. La spécification formelle de cette règle de réécriture selon le formalise défini (section 3.1) est comme suit :

$$Graph\_Transform(SEQ\bullet[n_i,n_k],SEQ\bullet[n_k,n_l])= SEQ\bullet[n_i,SEQ\bullet[n_k,n_l]]$$

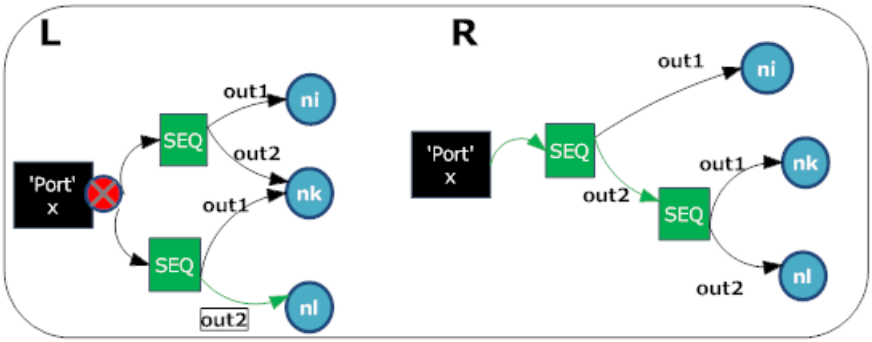


FIGURE 4.6 – Exemple de règles de réécriture de deux connecteurs de SEQ

4.1.3.2 Règle $R2$: AND et OR

Cette règle est appliquée dans le cas d'une interférence entre les deux connecteurs AND et OR. Le sous-graphe L de la figure 4.7 montre l'accès concurrent au port x (la variable x remplace l'identifiant du port). Chaque connecteur exprime une logique de déclenchement pour l'appel à ce port. La résolution de ce cas d'interférence revient à retrouver la logique de déclenchement du port en question. Le premier connecteur AND exige l'activation de ses deux ports d'entrées alors que le deuxième connecteur OR a seulement besoin de l'activation d'une seule entrée pour transmettre l'appel. Nous avons vu que plusieurs stratégies peuvent être choisies pour exprimer le résultat souhaité. La représentation formelle de cette règle de transformation est définit comme suit :

$Graph\_Transform([n_i,n_k]\bullet AND,[n_k,n_l]\bullet OR)= [n_k,n_l]\bullet OR$

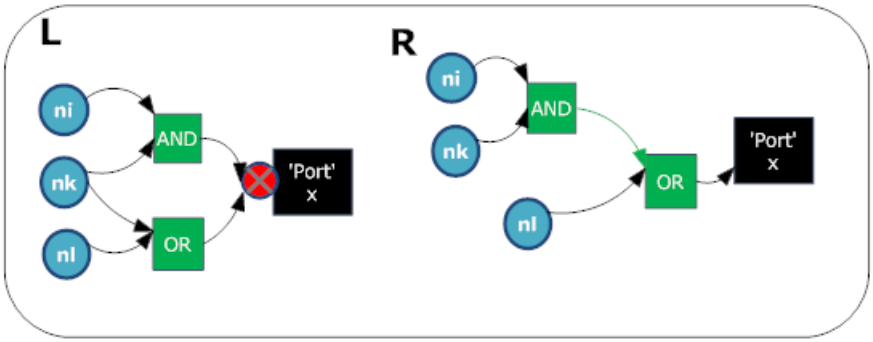


FIGURE 4.7 – Règle de réécriture *AND* et *OR* selon une stratégie restrictive

Pour une stratégie *restrictive*, qui respecte le comportement de chacun de deux connecteurs, la règle est donnée par la figure 4.7. La logique de déclenchement obtenue est la suivante : il faut avoir l'activation de deux nœuds d'entrée du $[n_i,n_k]\bullet AND$, ou bien la deuxième entrée du connecteur *OR* (n_l).

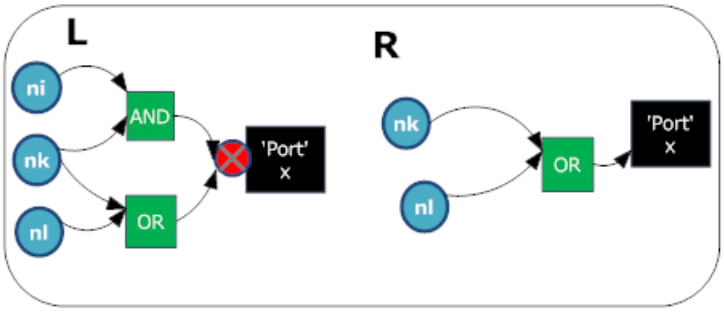


FIGURE 4.8 – Règle de réécriture *AND* et *OR* selon une stratégie permissive

Ce problème d'interférence pourra être résolution en utilisant une stratégie *permissive*. La règle de réécriture selon cette stratégie est donnée par la figure 4.8. Le résultat produit favorise le comportement apporté par le connecteur *OR*. Quelque soit l'état des évènements d'entrée du connecteur *AND*, avoir reçu l'un des évènements d'entrée du connecteur *OR* déclenchera l'appel du port de sortie.

4.1.3.3 Règle *R*3 : *PAR* et *AND*

La règle de transformation donnée par la figure 4.9 illustre un exemple d'interférence entre deux connecteurs appartenant à deux classes différentes. Le sous-graphe *R* montre la solution qui a été définie à la conception du mécanisme de composition. La formalisation de cette règle est définie par : $Graph\_Transform([n_l,x]\bullet AND, PAR\bullet[n_i,n_k])=x\bullet PAR\bullet[n_i,[.,n_l]\bullet AND\bullet n_k]$

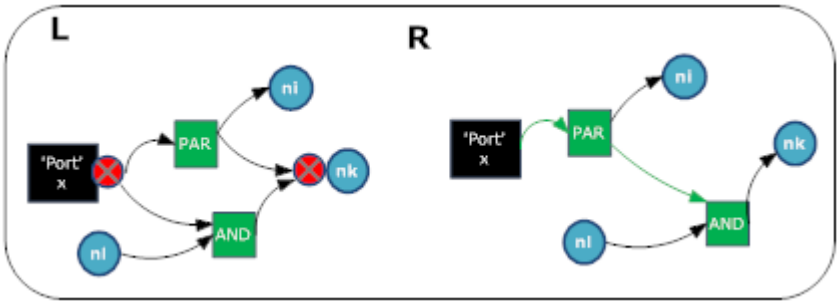


FIGURE 4.9 – Un exemple de règles de réécriture de deux connecteurs appartenant à deux classes différentes

4.1.4 Tâche 4 : Prouver des propriétés

L'ensemble des propriétés que le mécanisme de composition doit vérifier dépend du cadre dans lequel il va être appliqué. Dans notre cadre d'étude, notre mécanisme va composer des applications qui évoluent dans un environnement imprévisible avec une multitude d'entités. Cela implique que l'ensemble des entités d'adaptation à appliquer est déterminé en fonction de la disponibilité des dispositifs dans l'environnement. De plus, des entités d'adaptation peuvent alors être ajoutées ou retirées dynamiquement. L'ordre dans lequel ces entités vont être appliquées n'est pas connu à l'avance. Il n'est pas possible non plus de connaître a priori quelles sont toutes les entités à composer. Cependant le résultat de la composition peut être influencé par cet ordre. Pour toutes ces raisons, le mécanisme de composition doit être *symétrique* et *déterministe* ce qui permet de s'abstraire de l'ordre dans lequel les entités d'adaptation vont être déclenchées. La validation de ces propriétés pendant la phase de conception permettra de les avoir aussi à l'exécution.

Pour que le résultat du composition de plusieurs entités d'adaptation soit le même indépendamment de leur ordre de composition, il est nécessaire que cette opération de composition soit symétrique. La propriété de symétrie est une propriété logique qui est décomposée en trois sous propriétés qui sont : l'idempotence, la commutativité et l'associativité. Elle se définit comme suit :

Définition 8 : Symétrie
Soit ϕ *la fonction de composition avec* $\phi : \mathcal{M}_G \rightarrow \mathcal{M}_G$
Idempotence : $\forall g_1 \in \mathcal{M}_G \; \phi(g_1, g_1) = g1$
Commutativité $\forall g1, g2 \in \mathcal{M}_G \; \phi(g1, g2) = \phi(g2, g1)$
Associativité $\forall g1, g2, g3 \in \mathcal{M}_G \; \phi(\phi(g1, g2), g3) = \phi(g1, \phi(g2, g3))$

L'idempotence garantit que même si on applique plusieurs fois une instance d'entité d'adaptation, alors on a le même résultat qui est l'instance elle même appliquée une seule fois (par exemple, la composition de deux liens entre deux ports produira un seul lien). La propriété de commutativité permet de garantir que le résultat de la composition de deux instances d'entités d'adaptation donnera le même résultat quelque soit leur ordre de composition. Enfin, l'associativité permet de regrouper des compositions d'entités sous la forme d'une même opération. Cette propriété est utilisée pour garantir la symétrie dans le cas d'interférences d'ordre $p > 2$. En effet, elle va permettre de réutiliser le résultat d'une composition déjà réalisée.

Pour garantir cette propriété un ensemble de contraintes sont à respecter lors de la spécification des entités d'adaptation (les aspects). Le super designer construit donc en amont l'ensemble des contraintes :

- C1 : les points de jonctions ne peuvent être que de type B (*BlackBoxEntity*). Donc un nœud ajouté par une entité d'adaptation ne peut pas être un point de jonction pour une autre entité.
- C2 : la partie point de coupe d'une entité d'adaptation (aspect) n'exprime que la présence des entités logicielles et pas leur absence.
- C3 : les règles dans le greffon d'un aspect ne peuvent pas supprimer des nœuds de type B.
- C4 : tous les points de coupe des aspects sont évalués en utilisant le graphe de l'application initiale (avant l'application des aspects).

Cet ensemble des contraintes pour la propriété de symétrie réduit les interactions possibles entre les entités d'adaptation donc les types d'interférences que nous pouvons avoir. Par exemple, grâce à la contrainte C1, une entité d'adaptation ne pourra pas activer ou bien désactiver l'application d'une autre. De ce faite, il n'y a pas le besoin de la spécification d'un ordre entre les entités (il n'y a pas les problèmes de conflit ni de dépendance). Le non respect de l'une de ces contraintes a pour effet la perte de la propriété de symétrie. Par exemple si on permet au niveau d'un point de coupe d'un aspect d'exprimer la présence ou bien l'absence d'autres aspect, cela aurait pour conséquence d'introduire un ordre entre les aspects et entraînerait ainsi la perte de la propriété de commutativité de la composition.

La propriété de symétrie garantit par conséquent le déterminisme de l'opération de la composition qui est défini comme suit :

Définition 9 : Déterminisme
Soit un ensemble d'entités d'adaptation SQ_0, ... ;SQ_n et une composition d'application initiale G_{init}. L'application de toutes ces entités d'adaptation à G_{init} donnera le même résultat final.

La preuve de la propriété de symétrie se fait indépendamment de l'ensemble de composition à spécifier et de toute composition initiale d'application [FBALT+12b]. Cette propriété devra être maintenue par toutes les étapes qui forment le processus de la composition. Essentiellement, elle devra être vérifiée pour le mécanisme de résolution des interférences car ce dernier est le cœur de la composition. Il faudra s'assurer que l'ensemble des règles de réécriture appliquées durant la résolution garantit aussi cette propriété. Le détail de cette preuve est fournies dans la section A.2 de l'Annexe A .

L'ensemble de connecteurs pour lesquels les propriétés ont été prouvées, constitue le DSL (Domaine Specific Language) qui sera utilisé dans la spécification des entités d'adaptation par les *Designers*. Les règles de réécriture de graphe correspondant à ces connecteurs constituent la base de règles utilisée par le mécanisme de composition pour résoudre les problèmes d'interférences pendant l'exécution (figure 4.1).

↪ **À retenir.** *La propriété de symétrie permet de ne pas avoir à mettre en place un mécanisme qui doive faire la distinction entre les entités d'adaptation. Ces dernières peuvent donc être appliquées dans n'importe quel ordre et, par conséquent, être composées de manière imprévisible. Les*

interférences entre les entités d'adaptation sont gérées à partir d'un ensemble de règles de réécriture de graphe prédéfinies.

Jusqu'à présent nous avons détaillé toutes les étapes de conception de mécanisme de composition. Une fois validé, notre mécanisme interviendra pendant l'exécution pour la composition des applications en incluant l'identification ainsi que la résolution automatique des interférences apparues.

4.2 La composition dynamique et extensibilité

L'environnement dans lequel nos applications s'exécutent change fréquemment. Nos applications doivent alors être composées dynamiquement en y intégrant de nouvelles entités logicielles se trouvant dans son infrastructure logicielle. Ces entités logicielles ne peuvent pas être connues a priori. Nous avons vu dans le chapitre 2 que les entités d'adaptation qui décrivent la composition de ces entités logicielles doivent être gérées comme des préoccupations transverses [DL$^+$06]. Cela a pour objectif d'augmenter leur réutilisation et de réduire le nombre d'entités d'adaptation à écrire tout en garantissant leur indépendance. Ces entités d'adaptation sont alors l'entrée de notre mécanisme de composition [FBBLT$^+$11] qui doit les composer de manière déterministe.

En IAm, les entités d'adaptation qui vont être utilisées dans la construction du système ne sont pas connues a priori. De ce fait, le mécanisme de composition devra être capable de supporter *l'extension de cet ensemble pendent l'exécution*. C'est-à-dire qu'il devra être possible d'ajouter de nouvelles entités d'adaptation sans avoir besoin d'arrêter ni de redémarrer le mécanisme de composition. L'ajout des entités d'adaptation ne nécessite pas non plus de se préoccuper de leur composition avec les autres entités déjà présentes. Ceci est rendu possible grâce au mécanisme de gestion des interférences qui fait partie du processus de composition. Ce mécanisme garantit l'indépendance entre les entités d'adaptation et permet de produire la composition finale d'une application en résolvant tous les problèmes d'interférences apparus au moment de la composition. Notre mécanisme de composition effectue son raisonnement à partir d'une représentation abstraite de l'application en cours d'exécution.

Cette section fournit une description détaillée du déroulement du mécanisme de composition. Nous exposons les deux cas possibles pour la gestion des interférences : (1) une résolution en s'appuyant sur des règles de réécriture prédéfinies et (2) l'extension du mécanisme de résolution par une modélisation comportementale des nouveaux connecteurs.

Avant d'entamer le détail du processus de la composition, nous commençons par la description de entités d'adaptation qui sont l'entrée de ce mécanisme.

4.2.1 Les entités d'adaptation

Nous avons montré dans la section 1.3.2 que la Programmation Orientée Aspects (AOP) répond au mieux au besoin d'indépendance de spécification entre les entités d'adaptation. Dans nos travaux un aspect exprime une adaptation d'une application en agissant sur sa structure. Nous utilisons la partie de point de coupe pour identifier les entités logicielles qui font l'objet d'une adaptation. Ils permettent alors d'identifier les points de jonctions sur lesquels les greffons vont être tissés. Un greffon décrit l'adaptation effective. Les instructions élémentaires permises dans cette partie sont :

- Instanciation d'entités : Un greffon peut exprimer l'instanciation d'une entité logicielle de type *BlackBoxEntity* ainsi que des connecteurs. À l'issu de ce type d'instruction un nœud sera ajouté dans le graphe de l'application. L'ensemble des connecteurs définis par le Super Designer est utilisé par les designer pour décrire les opérations élémentaires de composition des entités logicielles.
- Interactions entre les entités : une interaction lie deux ports : source et destination. Ces ports correspondent aux entités logicielles sélectionnées par la partie point de coupe ou bien celles instanciées dans le greffon.

Nous notons que dans un greffon il n'est pas permis d'exprimer la suppression des entités logicielles *BlackBoxEntity* car ceci va à l'encontre de la propriété de symétrie. En effet, la suppression d'une entité par un aspect peut désactiver l'application d'un autre aspect (l'entité supprimée par un aspect peut être un point de jonction pour un autre). Une entité logicielle est supprimée donc seulement si elle n'est plus disponible dans l'environnement (disparition d'un dispositif, panne, etc.) ou bien lorsque l'aspect qui l'a instancié a été détissé.

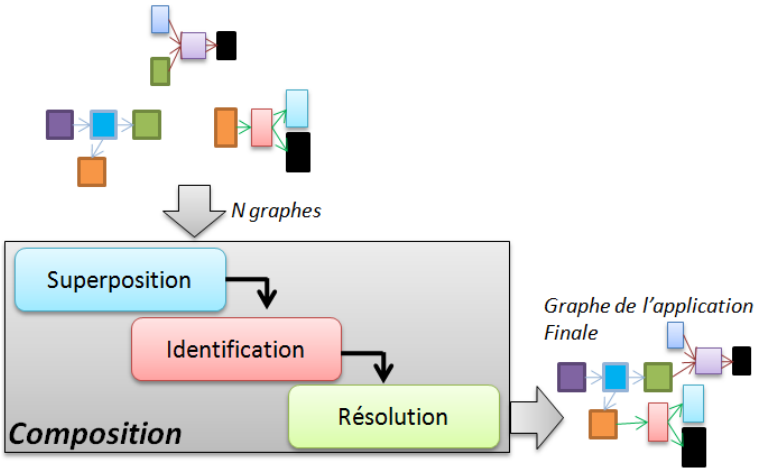


FIGURE 4.10 – Composition des entités d'adaptation

Un aspect est tissé si sa partie point de coupe est vérifiée. Le tissage d'un aspect sur une application de base donne naissance à un ou plusieurs graphes dont chacun correspond à une instance de greffon (il existe une opération de transformation des entités vers les graphes, détail section 4.2.5). Notre mécanisme de composition interviendra sur ces graphes superposés à l'application de base. Le mécanisme général de composition est donné par la figure 4.10.

Les sous-graphes sont superposés dans le but de produire un graphe final qui est l'union de ces sous-graphes (section 4.2.2). Nous obtenons donc G_{Global} qui représente l'application de tous les aspects qui ont été sélectionnés. L'étape suivante est l'identification des interférences (section 4.2.3). Le mécanisme de résolution des interférences (section 4.2.4) intervient aux endroits identifiés. La présentation détaillée de chacune de ces étapes fait l'objet des sections suivantes.

4.2.2 Superposition des graphes de composition

La première étape dans le processus de composition est la superposition des graphes produits par le tissage des aspects avec celui de l'application initiale G_{init}. Cette fonction prend en entrée un ensemble de graphes et le graphe G_{init} et produit comme résultat un graphe G_T. L'opération effectuée par la superposition est l'union des ensembles. Tous les nœuds possédants le même identificateur (c'est le cas où n_i appartient à B) sont fusionnés en un seul nœud en regroupant leurs arcs sortants et leurs arcs entrants.

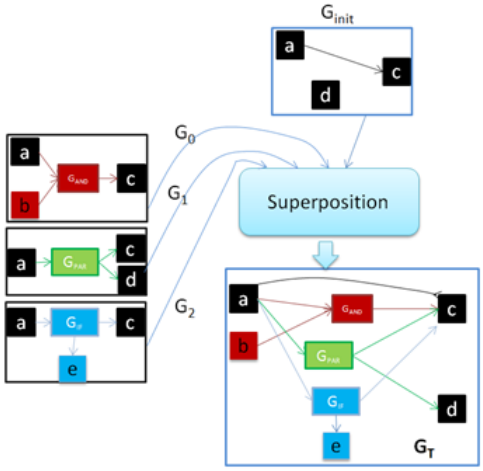


FIGURE 4.11 – Un exemple de superposition des graphes

La figure 4.11 présente un exemple de superposition. Nous avons attribué des couleurs différentes pour les nœuds et les arcs de chacun des graphes à superposer. Nous avons simplifié les graphes en inscrivant seulement l'identificateur du nœud (pas le type). Les nœuds partagés entre les graphes sont seulement les nœuds représentant les ports des entités "boîte noire". Ce processus peut être décrit plus formellement par l'algorithme 1.

Algorithm 1 $Superposition(ListG, Ginit) : GT$

y : nombre de graphes dans la liste $ListG$
A_{init} : l'ensemble des arc du graphe G_{init}
N_{init} : l'ensemble des nœuds du graphe G_{init}

for $i = 0$ to y **do**
  if n_i isNotIn N_{init} **then**
    Add n_i to N_{init}
  end if
  Add $inArc(n_i)$ to A_{init}
  Add $outArc(n_i)$ to A_{init}
end for

4.2.3 Identification des problèmes

La superposition des sous-graphes de composition avec le graphe de la composition initial peut faire apparaître des problèmes d'interférences. L'objectif de cette phase est d'identifier dans le graphe G_T ces endroits et de les marquer par un nœud spécial ($\otimes$). Il s'agit de chercher dans le graphe G_T les deux patrons d'interférences que nous avons définis précédemment. Il est alors nécessaire de parcourir les nœuds de notre graphe G_T pour détecter ces problèmes. Pour une interférence d'ordre p (c'est à dire un nœud qui possède p arcs sortants ou entrants), on ajoute $p-1$ nœuds pour marquer ces endroits.

Algorithm 2 InterferenceDetection(Graph G_T)

y : nombre de graphes dans la liste $ListG$

$P_e \subset N_T$
$P_s \subset N_T$
for $k = 0$ to $|P_s|$ **do**
  for $j = 0$ to $j < deg^+(n_k) - 1$ **do**
    $Add \otimes$ Node
  end for
end for
for $k = 0$ to $|P_e|$ **do**
  for j=0 $j < deg^-(n_k) - 1$ **do**
    $Add \otimes$ Node
  end for
end for

Cette opération prend en entrée le graphe $G_T(N_T, A_T)$ auquel elle ajoute des nœuds $\otimes$ donc l'ensemble de nœuds du graphe résultat N_T va être $N_T = N_T \cup \{\otimes_0, ..., \otimes_k\}$. L'ensemble A_T des arcs du graphe résultat de cette opération va être également modifié : des liens sont modifiés pour avoir comme destination (ou source) un nœud de $\otimes$ et des nouveaux arcs sont créés (qui ont comme source ou destination le nœud de type $\otimes$). L'algorithme 2 montre le processus de détection des interférences.

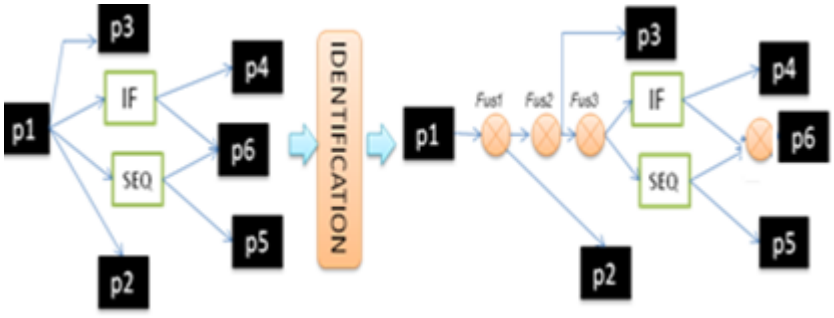


FIGURE 4.12 – L'étape de détection des interférences marque les endroits par un nœud spéciale $\otimes$

La figure 4.12 illustre le résultat de l'application de cet algorithme sur le graphe résultant de l'étape de la superposition. Cet exemple illustre quatre interférences : trois interférences à la sortie

du port $p1$ et une interférence à l'entrée du port $p6$. Le processus de détection d'interférence a ajouté quatre nœuds $\otimes$. Il n'y a pas d'importance quant à l'ordre dans lequel les nœuds de fusion sont ajoutés (on peut avoir un nœud pour fusionner IF avec $p3$ ou bien $p2$ à la place de SEQ ; le résultat reste le même). Ceci est rendu possible grâce à la propriété de symétrie.

4.2.4 La résolution

L'objectif de cette phase est d'effectuer une réécriture du graphe où les interférences ont été marquées. Les règles de réécriture nécessaire pour la résolution des problèmes sont déjà fournies pour les connecteurs de base. Toutefois nous avons évoqué précédemment que les designer pourront utiliser des connecteurs en dehors de cette liste prédéfinies sous condition qu'ils fournissent la description comportementale des nouveaux connecteurs. Dans la section suivante, nous détaillons tout d'abord le mécanisme de résolution classique (défini statiquement) et ensuite nous présentons le cas d'extension du mécanisme de résolution pour considérer de nouveaux connecteurs.

4.2.4.1 Résolution classique par réutilisation des règles prédéfinies

L'objectif de cette étape est de gérer les interférences se produisant lors de la composition de plusieurs graphes représentants des instances des entités d'adaptation. Cette résolution est conforme à la logique de résolution qui a été définie statiquement. C'est à partir des nœuds $\otimes$ (ajoutés durant l'étape de détection) que le mécanisme de gestion des interférences construira sa solution. Ces nœuds $\otimes$ seront remplacer par des nœuds de type $WhiteBoxEntity$. Notre mécanisme mettra en ouvre la logique qui a été définie par le Super-Designer pendant la phase de conception. Donc le comportement mis en place par cette résolution va être consistant avec la logique définie en amont. Basiquement, l'algorithme pour résoudre ces interférences parcourt l'ensemble de tous les nœuds $\otimes$ dans l'objectif de faire tourner sur chacun d'eux le moteur de transformation de graphe. Ce processus peut être décrit plus formellement par l'algorithme 3.

Algorithm 3 InterferenceResolution (Graph G)

LFus est la liste de nœud de type $\otimes$
k : le nombre d'élément dans la liste LFus

for m = 0 to k **do**
  if $OUTArc(\otimes_m) = 2$ **then**
    Node n_i= sucesseurs($\otimes_m$)
    Node n_j= sucesseurs($\otimes_m$)
    $Rewriting(n_i, n_j)$
  else
    if $INArc(\otimes_m) = 2$ **then**
      Node n_i= $predecesseurs(\otimes_m)$
      Node n_j= $predecesseurs$ $(\otimes_m)$
      $Rewriting(n_i, n_j)$
    end if
  end if
end for

La fonction *Rewriting* prend comme entrée les deux nœuds successeurs au nœud $\otimes$ (respectivement les deux nœuds prédécesseurs) et propose de les connecter autrement à travers l'application d'une règle de réécriture de graphe. Ceci est rendu possible grâce à la connaissance de la sémantique des connecteurs. En effet, si nous avons plusieurs connecteurs à la sortie d'un nœud (respectivement à l'entrée d'un nœud), le mécanisme de réécriture produira un connecteur final à partir duquel nous retrouvons tous les comportements spécifiés par l'ensemble de connecteurs de départ (avant la résolution). Cet objectif a été fixé par le Super designer à travers les règles fournies. Ces dernières permettent de réécrire toutes les combinaisons possibles des connecteurs deux à deux. L'algorithme 4 décrit le déroulement de la fonction *Rewriting*.

Algorithm 4 Rewtriting (Node n_i, Node n_j)

Rule R=$SelectRule(type(n_i),type(n_j))$
if R is $NULL$ **then**
  $DefaultSolution()$
else
  $Apply(R)$
end if

Le choix de la règle de réécriture (*SelectRule*) à appliquer pour résoudre une interférence se base sur le type des nœuds passés en entrée. À chaque couple de connecteurs est associé au moins une règle de réécriture. Plusieurs règles peuvent être proposées décrivant différentes configurations de ces deux connecteurs (par exemple les connecteurs partagent un nœud, etc.). La règle sélectionnée sera par la suite appliquée par le moteur de transformation de graphe. L'application d'une règle de réécriture pourra résoudre le problème définitivement ou bien partiellement. Une résolution partielle propose la réécriture de deux connecteurs en propageant le nœud $\otimes$ vers les arcs sortant de ces nœuds. Cela ajoutera de nouveaux nœuds $\otimes$ et nécessitera par la suite l'application d'autres règles de réécriture (par exemple la règle de réécriture de deux IF illustrée par la figure A.4 dans Annexe A). Dans le cas où le Super Désigner n'a pas prévu une règle de réécriture pour résoudre le problème, une solution par défaut va être appliquée (*DefaultSolution*). C'est le cas par exemple lorsque un port possède deux liaisons n'utilisant pas de connecteurs et sont en conflits ; la solution par défaut consiste alors à ajouter un connecteur *PAR* entre les deux liaisons (règle A.1 dans Annexe A). Ceci garantit également la propriété de symétrie de l'opération de réécriture (la réécriture de chacun de ces connecteurs a été définie comme symétrique). L'application d'une règle de réécriture selon l'approche que nous avons choisit (SPO) nécessite la recherche du sous-graphe L dans G_T. La complexité de cette opération sera alors fortement liée à la complexité du moteur de transformation choisi par le Super Designer pendant la conception.

Une fois que les interférences de type $1-n$ ont été résolues, nous enchaînons par la résolution du deuxième type d'inférence (type $n-1$). Il s'agit des problèmes d'accès concurrents à un port d'une entité logicielle. La solution que nous avons proposée précédemment consiste à réécrire ces connecteurs pour ne garder qu'un seul appel à ce port. Chaque connecteur simple de type $n-1$ implémente une logique de déclenchement qui sera utilisée pour le calcul de la logique de déclenchement du connecteur complexe. Le principe de réécriture de graphe va être également utilisé dans cette résolution mais cette fois ci pour déterminer quelle logique de déclenchement doit être mise en ouvre (c'est à dire comment réécrire les connecteurs de base pour avoir la logique de déclenchement souhaitée).

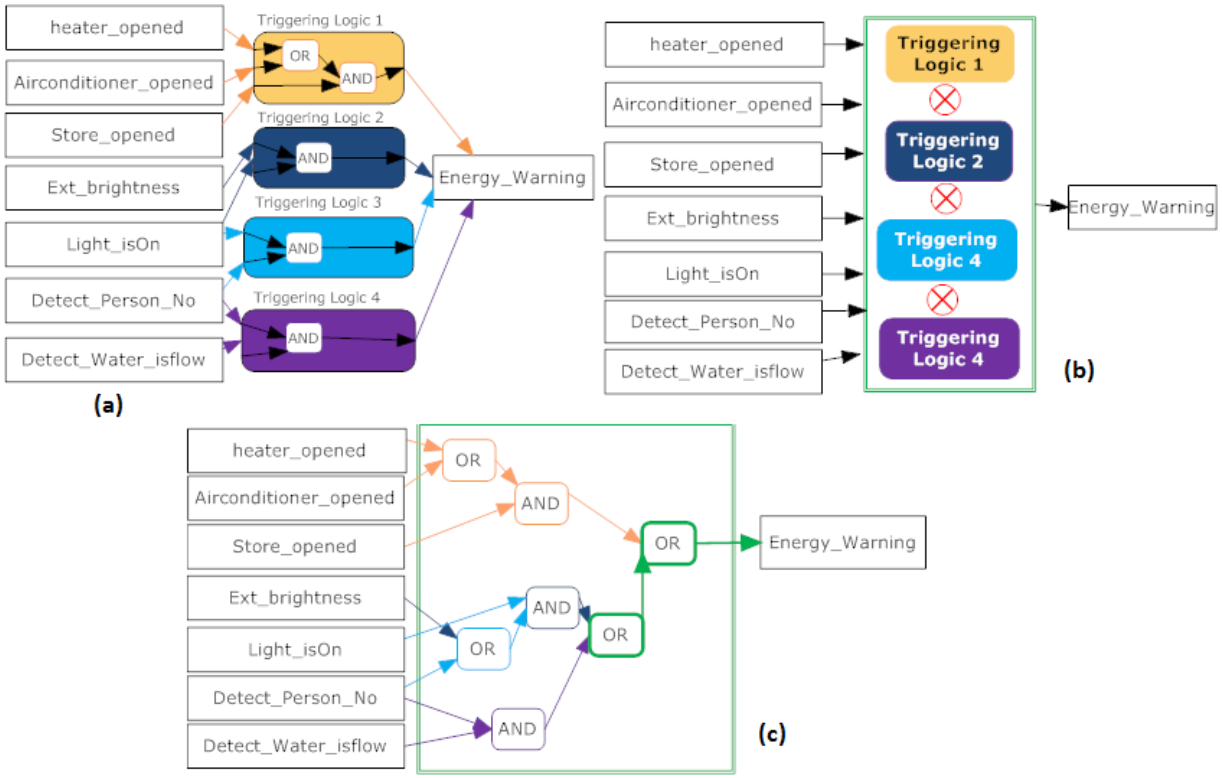


FIGURE 4.13 – Un exemple de résolution des interférences de type $n-1$

La figure 4.13.a illustre un exemple d'interférence de type $n-1$. Dans cet exemple, il y a 4 arcs entrants vers le port *Energy_Warning*. Il s'agit d'une interférence d'ordre 3. Chaque instance des entités d'adaptation exprime une logique de déclenchement de cette alarme. La réécriture de ces connecteurs produira le graphe (figure 4.13.c) qui définit la logique de déclenchement de ce port où il y avait un problème d'accès concurrent.

4.2.4.2 Résolution avec une modélisation comportementale des nouveaux connecteurs

Nos règles de réécriture expriment une nouvelle manière d'interconnecter les connecteurs dans le cas d'une interférence. L'utilisation d'un connecteur non connu a priori active un autre processus de résolution qui sera décrit dans cette section. Nous détaillons toutes les phases qui mènent à la génération d'une règle de réécriture pour tout nouveau connecteur et par conséquent l'extension du mécanisme de composition pour supporter ultérieurement l'utilisation de ce connecteur. Une fois que la règle de réécriture a été générée, le processus de résolution reste le même que celui qui a été détaillé dans la section 4.2.4.1.

L'algorithme 5 met en œuvre le processus d'extension du mécanisme de résolution des interférences qui a été décrit dans la section 3.3.3. La première opération est la composition des automates dont l'un parmi eux correspond à un nouveau connecteur. La fonction *AutomComposition* implémente une stratégie donnée de composition (synchrone, parallèle, etc.). L'automate obtenu sera l'entrée de la fonction *Transform* qui produira l'équation régissant le comportement de cet automate sous forme de combinaisons de connecteurs de base. Nous avons vu qu'une règle de

Algorithm 5 CompositionExtension (Node $NewC_i$, Node n_i)

$FusAutom$=$AutomComposition(autom(NewC_i), autom(n_j))$
Eq= $Transform(FusAutom)$
RHS= $EquationToGraph(Eq)$
$Rule$= $createRule(NewC_i$, Node n_i,$RHS)$
$addRule(Rule)$

réécriture est définie par (L,R). Le sous-graphe L correspond à l'interférence qui a été identifiée entre les connecteurs $NewC_i$ et n_i. Pour avoir une règles valide, il faudra compléter la partie R de la règle. Pour cela, la fonction *EquationToGraph* a comme paramètre l'équation Eq qui servira essentiellement à la génération du sous-graphe R. À ce stade, nous avons obtenu les deux parties de la règle qui sera créée par la fonction *createRule*. La règle obtenue *Rule* sera ajoutée à la base de règles du mécanisme qui sera par la suite capable de résoudre ce nouveau cas d'interférence.

Durant la phase de la conception statique du mécanisme de composition, nous avons évoqué que les règles de réécriture fournies par le Super-Designer devront être validées par rapport à des propriétés. L'algorithme que nous venons de détailler n'inclut pas explicitement cette phase de vérification. En effet, la fonction *AutomComposition* englobe cette phase de validation en faisant du model-checking [JGP00]. Cette approche nécessite un modèle du système à partir duquel la satisfaction des formules est vérifiée (ces formules représentent des propriétés requises). La machine de Mealy synchrone est un des modèles pertinent pour appliquer les techniques de model-checking. Ces formules peuvent être décrites à l'aide d'une logique formelle interprétées par des automates tel que *CTL* [JGP00]. Il s'agit d'un langage formel basé sur la logique du premier ordre où les assertions liées aux comportements attendus sont facilement exprimés. Nous nous intéressons particulièrement à la propriété de *symétrie* qui a été détaillée durant la phase de conception statique du mécanisme de composition. La composition d'automates parallèles ou synchronisés garantit par définition cette propriété ; il n'est donc pas nécessaire de la prouver de nouveau pour la règle de réécriture obtenue.

Jusqu'à présent nous avons détaillé toutes les opérations au niveau de la représentation abstraite qu'est le graphe. Ceci est rendu possible grâce à deux transformations : de l'application en cours d'exécution vers le graphe et du graphe vers l'application à l'exécution.

4.2.5 Composition et transformations

Nous avons défini le concept de composition dans un formalisme de graphe. Des transformations de et vers ce formalisme sont nécessaires : (1) Une transformation de l'application en cours d'exécution $\hookrightarrow$ modèle abstrait de l'application ; (2) transformation des entités d'adaptation $\hookrightarrow$ modèle abstrait de l'application (3) le mécanisme de composition réalise à l'exécution des transformations en interne qui sont homogènes ; (4) une transformation de la sortie sous forme de modèle abstrait de l'application $\hookrightarrow$ l'application s'exécutant. La figure4.14 illustre toutes ces transformations.

Phase 1 : Transformation application en cours d'exécution $\hookrightarrow$ modèle abstrait Chaque modification de l'application s'exécutant sera suivi par une mise à jour de son modèle abstrait. Deux stratégies de construction du modèle abstrait (graphe) peuvent être utilisées. La première stratégie part toujours de l'introspection de l'application pour identifier les changements et mettre à jour son

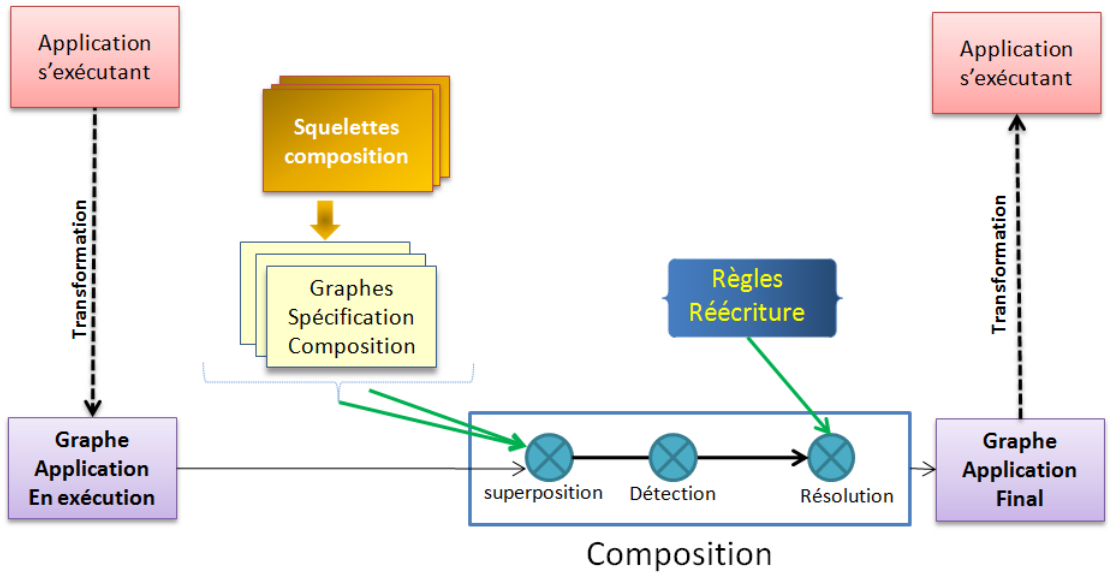


FIGURE 4.14 – Des transformations nécessaires avant, pendant et après le processus de composition

modèle abstrait. Cette stratégie ne convient pas en IAm [Fer11]. Le modèle peut alors être mis à jour de manière incrémentale, ce qui permet de faire évoluer en parallèle le modèle abstrait et l'application s'exécutant.

Phase 2 : Transformation entités d'adaptation ↪ modèle abstrait Les spécifications des compositions peuvent aussi être données dans un autre formalise que celui des graphes (par exemple écrites à l'aide d'un langage). Dans ce cas une autre transformation est nécessaire pour produire les sous-graphes de composition conformes à notre modèle. En effet, les règles décrites dans les entités d'adaptation sont génériques et travaillent sur les types d'entités logicielles. Cette transformation produit des instances des entités d'adaptation en remplaçant les types par les entités qui sont réellement présentes dans l'application. Le résultat de cette instantiation est conforme au modèle utilisé par le mécanisme de composition.

Phase 3 : Transformation endogène modèle abstrait ↪ modèle abstrait Le mécanisme de composition applique des transformations endogènes model-to-model [MVG06] paramétrées par les entités d'adaptation et les règles de réécriture. Plus précisément, le mécanisme de composition effectue deux transformations comme nous pouvons le voir en figure 4.14. La première transformation est paramétrée par les instances des entités d'adaptation. Elle a pour objectif de produire une unique instance de composition d'entités logicielles. La deuxième transformation est paramétrée par des règles de réécriture définies pour ce modèle abstrait. Elle a pour but de résoudre les éventuels problèmes d'interférence dans la composition obtenue par la première transformation. En cas d'absence de règles de réécriture, la transformation ne modifie pas le résultat de la superposition de ces entités (pas de résolution des interférences). Ces deux transformations garantissent la cohérence de l'application résultante.

Phase 4 : Transformation modèle abstrait ↪ application en cours d'exécution Enfin, lors de cette phase, lorsque le modèle abstrait est modifié par la composition, l'application s'exécutant est

mise à jour. Afin de ne modifier dans l'application que les parties qui ont changées, le modèle de l'application est comparé au modèle de l'application adaptée. Dans l'objectif de maintenir une application consistante, les opérations qui seront effectuées pour mettre à jour l'application doivent respecter cet ordre :

1. retraits d'interactions
2. retraits d'entités
3. ajout d'entités
4. ajout d'interactions
5. modification des propriétés d'entités

4.3 Conclusion

Ce chapitre a présenté la mise en œuvre du processus de la composition. Une étape importante dans ce processus est la gestion des interférences. L'approche a été définie à deux niveaux :

1. Définition manuelle et statique d'un mécanisme de résolution qui exploite la connaissance de la sémantique de connecteurs de base utilisés dans la spécifications de composition
2. Extension du mécanisme de résolution durant la phase de la composition dynamique pour supporter des nouveaux connecteurs. Cela est rendu possible grâce à l'utilisation d'une description comportementale de ces connecteurs.

Le chapitre suivant présente le détail de l'implémentation de notre approche en utilisant une plateforme adaptée aux applications en IAm.

Troisième partie

Validation et Conclusions

CHAPITRE 5

Mise en œuvre et Expérimentation

Sommaire

Dans ce chapitre, nous présentons l'implémentation des concepts que nous avons présentés précédemment : le modèle d'application et le modèle d'adaptation. À partir d'un scénario, que nous appliquerons au cadre de la domotique, nous proposons une mise en oeuvre de notre mécanisme de composition qui prendra en entrée une application basée sur une infrastructure logicielle de services pour dispositifs et des spécifications d'adaptations. Pour cette mise en œuvre, nous utiliserons la plateforme d'exécution à base de composants WComp.

5.1 Modèle d'application SLCA

Les caractéristiques de l'IAm exposés dans le chapitre 1, lorsqu'elles sont combinées les unes avec les autres, sont à l'origine des choix effectués pour l'implémentation de notre solution. Dans cette section, nous détaillons le modèle d'application utilisé dans le cadre de l'informatique ambiante. Ce modèle permet de créer des applications dynamiques à partir d'une infrastructure de services basés sur des communications par évènements [Hou10]. Ce modèle vise à prendre en compte les différentes problématiques pour l'adaptation des applications d'informatique ambiante. Les approches orientées service ont apporté des solutions aux problèmes de l'*hétérogénéité* et l'*interopérabilité*. L'interopérabilité est apportée par les services web [PTDL07] au niveau des protocoles de communication en reposant sur des technologies web largement utilisées tels que le protocole HTTP et le langage XML. Ces technologies ont permis de créer des applications à partir de services s'exécutant avec des langages de programmation hétérogènes, et des architectures matérielles différentes. Fournisseurs et consommateurs peuvent utiliser des architectures matérielles et/ou logicielles différentes. Un autre enjeu se

réfère à la manière de *gérer l'imprévisibilité* de disponibilité de ces entités pour pouvoir les intégrer et les retirer facilement du système ambiant. Les composants sont beaucoup plus faciles à manipuler que les services puisqu'ils s'exécutent dans un environnement contrôlé et peuvent être instanciés et manipulés dynamiquement. Pour bénéficier des avantages des différents paradigmes, il existe des modèles d'application multi-paradigmes qui combinent à la fois services et composants. Nous utilisons le modèle *SLCA (Service Lightweight Component Architecture)* [CFW09] qui a pour but de créer des applications basées sur les dispositifs présents dans l'environnement en utilisant une composition dynamique de services pour dispositif. SLCA définit un modèle d'architecture compositionnelle basé sur trois paradigmes : *composant*, *service* et *événement* :

- *Assemblage de composants*. Un service web composite est créé à partir d'un assemblage de composants.
- *Architecture orientée service*. Une application est un graphe de service web et de service web composites.
- *Communication événementielle*. L'utilisation ce type de communication augmente le découplage entre les entités et favorise donc la dynamicité [Hou10].

5.1.1 Les éléments du modèle

5.1.1.1 Assemblage de composants légers

Pour concevoir des services Web pour dispositif composites, le modèle SLCA utilise des composants légers selon le modèle LCA (Lightweight Component Architecture). C'est un modèle inspiré de Beans [Eng97]. Un composant se définit par un ensemble de ports d'entrées, de sorties et des propriétés. Ces composants sont dits légers car ils sont rattachés à un même processus et ils s'exécutent dans un même espace de mémoire. Ceci va impacter leurs interactions qui deviennent plus simples et plus efficaces. Ces composants respectent le concept de boîte noire ; ils ne contiennent pas de référence entre eux lors de la conception. Ces composants sont dits légers dans le sens où ils n'embarquent pas de services techniques inutiles dans un environnement local. La gestion des événements et des propriétés sont les seuls codes non-fonctionnels qui doivent être présents dans le code des composants.

Dans ce modèle, les composants possèdent une interface qui est un ensemble de ports d'entrée (des méthodes) et de ports de sortie (des événements). Chaque port a un identifiant unique et est typé par des paramètres. Les interactions entre les composants relient un port de sortie d'un composant à un ou plusieurs ports d'entrée de composants. Lorsque l'événement est envoyé par le composant qui est à la source de la liaison, le flot de contrôle passe alors aux composants destinataires dans un ordre indéterminé. Cet ordre pourra être fixé par l'utilisation de connecteurs tel que la séquence ce qui rend la gestion de flot de contrôle de l'application déterministe. Les composants sont décrits par des fichiers sauvegardés dans un répertoire (*Repository*). Ils sont instanciables dans une application à partir de ces fichiers. Cette liste de fichier peut être modifiée à l'exécution, c'est à dire charger et décharger des fichiers. Deux types de composants sont définis : les composant *proxies* et les composants *fonctionnels*. Lorsqu'un service représentant un dispositif est découvert, un composant spécifique appelé composant *proxy* peut être généré à partir de la description du service et instancié pour faire partie de l'assemblage de composants. Les composants proxies jouent le rôle d'interface entre les services de l'infrastructure et les composants d'un assemblage. La figure 5.1 illustre un service composite selon le modèle SLCA. Ce dernier propose un modèle d'architecture pour la composition de services basé sur des assemblages de composants légers.

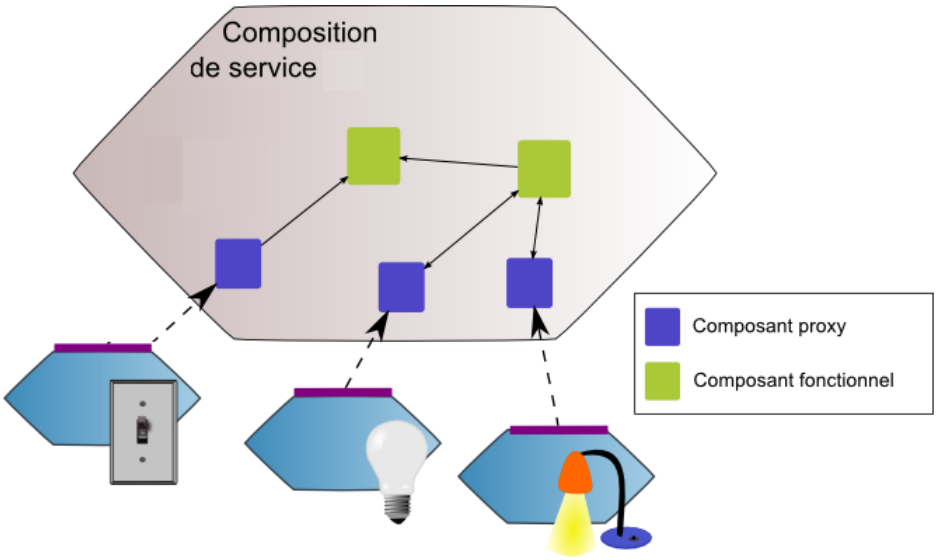


FIGURE 5.1 – Un service composition selon le modèle d'architecture SLCA

5.1.1.2 Service Composite SLCA

Le modèle SLCA construit les applications à partir des services de l'infrastructure découvrables dynamiquement qui représentent les dispositifs disponibles. Ces applications sont conçues par composition de services en assemblant des composants. Un service composite encapsule donc un assemblage de composants. Les composants proxies d'autres services composites et des services web peuvent faire partie de cet assemblage de composants. Cela va créer des applications à partir des dispositifs présents dans l'environnement ainsi que d'autres compositions de service. De ce fait, un service composite peut créer une application communiquant avec un autre service composite. La figure 5.2 montre qu'une application selon SLCA est alors un graphe de services dont chacun encapsule un assemblage de composants légers. L'apparition d'un dispositif va engendrer l'instantiation d'un composant proxy qui sera à son tour utilisé dans la composition interne d'un ou plusieurs services. Le graphe de l'application va alors subir cette variation et va être incrémenté.

Un service composite possède deux interfaces dynamiques : une interface *fonctionnelle* et une de *contrôle*. L'interface fonctionnelle permet d'exporter une application créée localement par l'assemblage de composants vers l'infrastructure de service (ceci permettra d'utiliser un service composite par d'autres). L'interface de contrôle permet d'agir dynamiquement sur l'assemblage interne de composants. Elle permet d'ajouter ou supprimer des composants et des interactions et de récupérer des informations sur l'assemblage (type de composants disponibles, liste des composants instanciés, liste des interactions, etc.). Ce modèle fournit donc les éléments nécessaires pour l'adaptation. En effet, une entité qui utilise un service composite à travers un proxy, pourra agir sur la structure de ce dernier. L'adaptation structurelle des services composites et des applications est donc possible dans le modèle par ses propres entités.

Le changement dans l'infrastructure (apparition ou disparition des dispositifs) se traduit par l'instantiation ou bien la suppression des composants proxies ainsi que leurs interactions. L'adaptation porte alors sur la composition interne d'un service composite ou bien une application locale LCA.

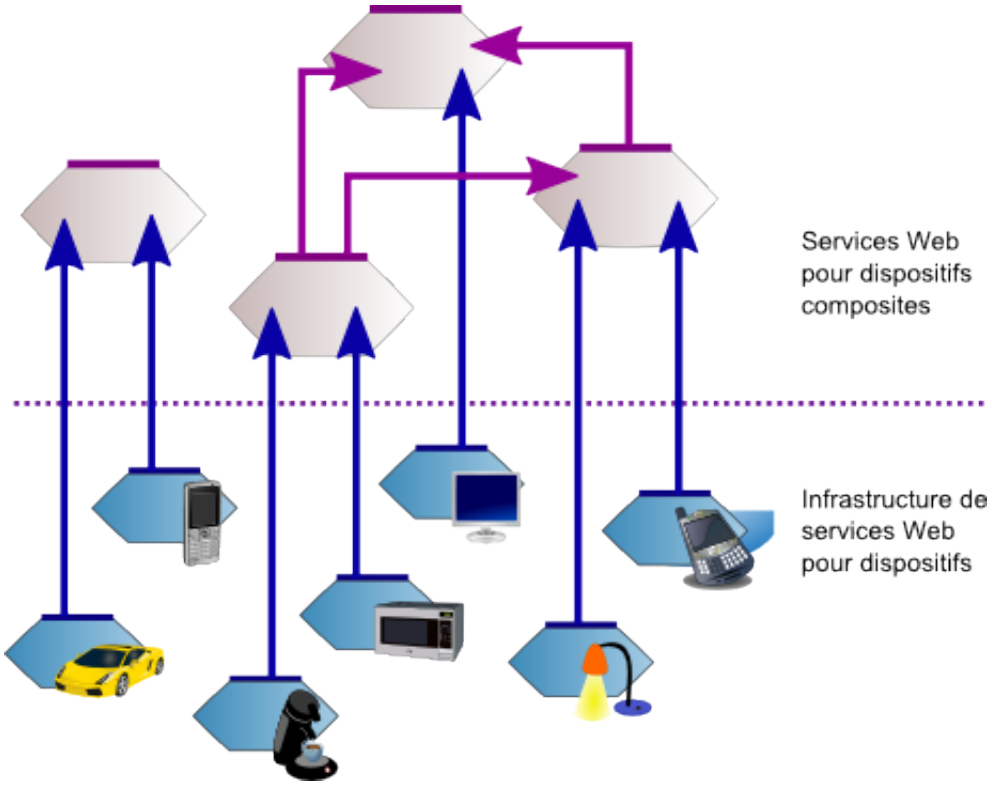


FIGURE 5.2 – Un graphe de composition de services SLCA

Dans la suite nous représentons à l'aide de notre modèle de graphe l'assemblage de composants d'un service composite (ou une application locale).

5.1.2 Représentation d'un service SLCA avec notre modèle de graphe

Nous avons présenté dans la section 3.1 notre modèle de graphe qui est une abstraction d'une application en cours d'exécution. Ce modèle a été défini indépendamment des outils de transformation de graphes. Pour sa mise en œvre, nous avons utilisé AGG (Attributed Graph Grammar) [Tae00]. C'est un environnement général pour la transformation algébrique des graphes, basé sur un concept d'attribution très flexible. Les objets d'un graphe selon AGG sont autorisés à être attribués par toute sorte d'objets Java. Les règle de réécriture de graphes peuvent être équipées de calculs arbitraires (décrits par une expression Java) sur ces objets Java. L'environnement AGG se compose d'une interface graphique comprenant plusieurs éditeurs visuels, et un interprète, qui est également disponible sous forme d'API Java.

Dans notre implémentation, nous considérons le modèle de graphe (appelé aussi *typed graph*) illustré par la figure 5.3. Les nœuds (la classe *Node*) sont une abstraction des entités impliquées dans une application. Ils peuvent être un nœud de sémantique connue (*connecteurs*) ou bien un nœud de sémantique inconnue (port de composants). Chaque nœud est spécifié par deux attributs : *CTy* est le type de nœud et *CN* est le nom de l'instance qui doit être unique. Comme exemple, nous définissons le type graphe d'une application à base de composants. Pour les nœuds de sémantique inconnue, nous représentons les ports des composants *CTy='Port'* et *CN* est sous la forme *Component_Name.PortName*.

Pour les nœuds de sémantique connue, le *CTy* spécifie le type du connecteurs(*SEQ*, *AND*, etc.).

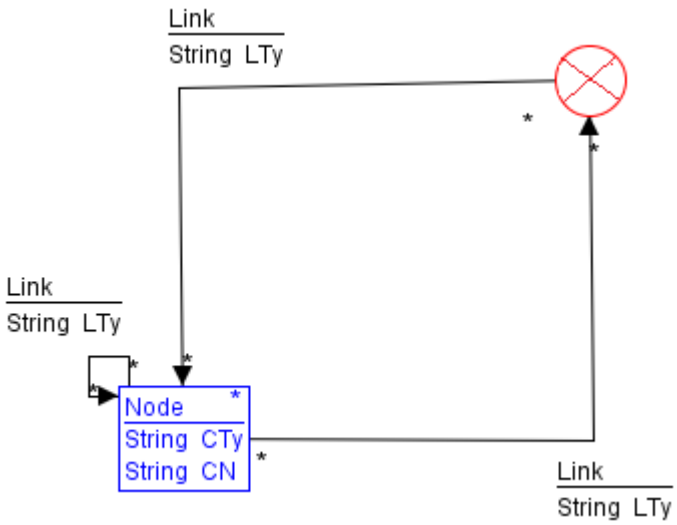


FIGURE 5.3 – Implémentation du modèle de graphe sous AGG

Les arcs du graphe (classe *Link* dans la figure 5.3) modélisent les interactions entre les entités impliquées dans nos applications. Chaque arc modélise une liaison port à port. Il est étiqueté par une chaîne de caractères dont la signification est liée à la sémantique du nœud source. Par exemple nous avons vu précédemment que le connecteur *SEQ* exécute deux actions dans un ordre défini. Pour spécifier cet ordre au niveau du graphe des étiquettes *«out1»* et *«out2»* sont ajoutées au niveau des arcs marquant cet ordre. L'étiquette par défaut est *«L»*, c'est à dire il n'y a pas de sémantique spécifique pour cet arc. Le tableau 5.1 présente l'ensemble des étiquettes utilisées dans notre implémentation.

TABLE 5.1 – Ensemble d'étiquettes pour les arcs.

Type de Nœud	Étiquettes
SEQ	out1, out2
IF	T, F, C

Notre *typed graph* définit une autre classe de nœud : $\otimes$. Ces nœuds sont utilisés pour marquer dans le graphe les endroits où une interférences a été détectée. La figure 5.4 est une instance d'un graphe selon le modèle défini. L'émission de l'événement SetOn du composant *Switch1* déclenche deux actions en séquence : *light1.SetTarget* ensuite *light2.setTarget*.

Nos graphes sont obtenus à l'issu de l'application d'un ou plusieurs entités d'adaptation. Dans la section suivante nous présentons la technique utilisée pour la spécification de ces entités d'adaptation (les aspects d'assemblage). Nous présentons aussi la transformation de ces entités d'adaptation vers un graphe à partir duquel nous appliquons notre raisonneur pour la gestion des problèmes d'interférences.

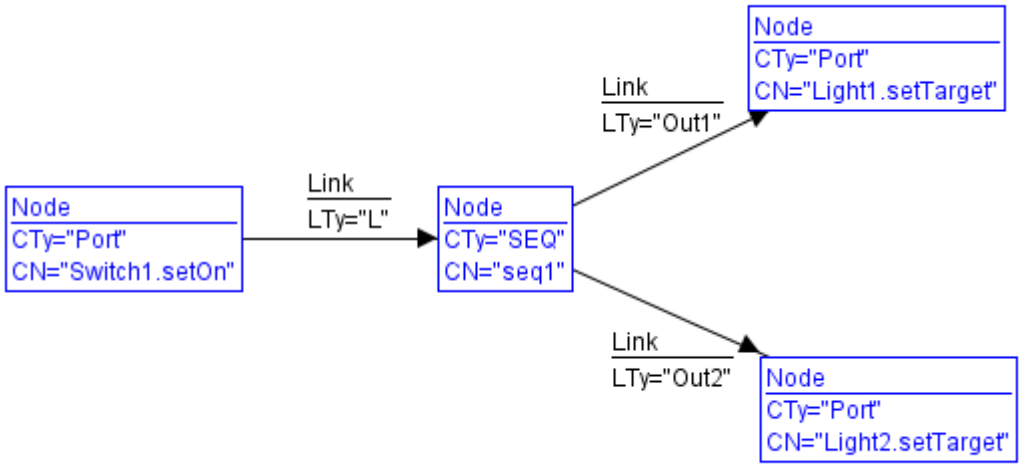


FIGURE 5.4 – Un exemple de graphe de composition en utilisant le modèle défini sous AGG

5.2 Les Aspects d'Assemblage (AA)

5.2.1 Vocation des AA et auto-adaptation au contexte

Au cours de sa vie, une application en IAm doit évoluer dans des environnements très variés que le développeur ne peut prévoir statiquement à la conception. Il faut donc qu'à l'exécution le système puisse s'adapter à ces changements. Nous avons vu (section 1.2) qu'il n'est pas envisageable de supposer l'existence d'un développeur derrière toutes les applications qui se trouvent dans ces conditions de variabilité. Le système devra avoir la capacité de s'adapter automatiquement (auto-adaptation).

L'équipe Rainbow a proposé dans ses précédents travaux un mécanisme d'auto-adaptation *compositionelle* basé sur les *Aspects d'Assemblage* (AA) [TLR+12]. Les aspects d'assemblage sont un modèle de spécification d'adaptation basés sur la programmation orientée aspect (AOP). Ce modèle est *assez générique* ce qui lui a permis d'être utilisé pour spécifier différentes préoccupations dans divers domaines. Il est suffisamment abstrait pour pouvoir être *indépendant des plates-formes*. Contrairement à un aspect classique qui permet de modifier le code d'un programme, un AA exprime sous forme de règles une adaptation des applications en agissant sur leurs structures. Il modifie soit la structure d'un assemblage de composants représentant une application locale (LCA), soit la structure d'un assemblage encapsulé dans un service composite (SLCA). L'utilisation des AA nous a permis d'exprimer nos règles comme des adaptations *indépendantes*.

La spécification d'un AA n'implique pas forcément sa participation immédiate dans le processus d'adaptation des applications. Un AA passe par différents états qui forment son cycle de vie (figure 5.5). Pour qu'il soit reconnu comme une adaptation potentielle, un AA devra être sélectionné explicitement. La sélection d'un AA signifie qu'il fait partie de la logique d'adaptation courante (deux mécanismes de sélection : automatique et manuelle). Cette sélection n'est pas non plus suffisante pour qu'un AA soit appliqué. En effet, les AAs sont déclenchés en réaction à des événements provenant

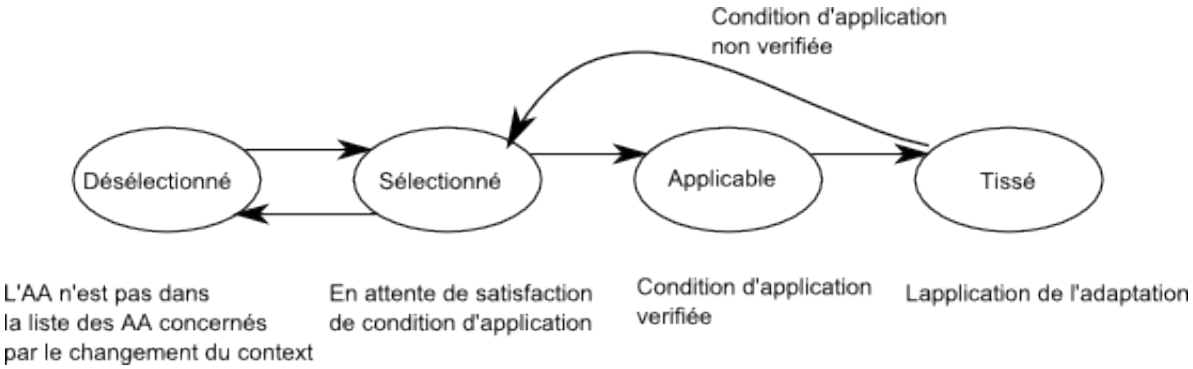


FIGURE 5.5 – Cycle de vie d'un Aspect d'Assemblage

de l'infrastructure logicielle. Le déclencheur du processus d'adaptation est donc externe à l'application et porte sur des variations dans le contexte d'exécution d'une application. Une variation est un élément imprévu qui pourra être : apparition/disparition d'éléments dans l'application (proxies vers des services), ajout/retrait des AAs. Ces variations ont pour effet de déclencher l'évaluation des AA et mettre à jour leur état courant (rester dans le même état ou basculer vers un autre parmi ceux de la figure 5.5). Un AA doit être appliqué lorsque l'infrastructure logicielle le permet et inversement doit être annulé lorsqu'une fonctionnalité nécessaire à son application est retirée. Il est donc appliqué si une condition, qui porte sur les dispositifs présents dans l'infrastructure, est vérifiée. Seulement dans un tel cas, l'AA sera tissé et l'adaptation sera effective.

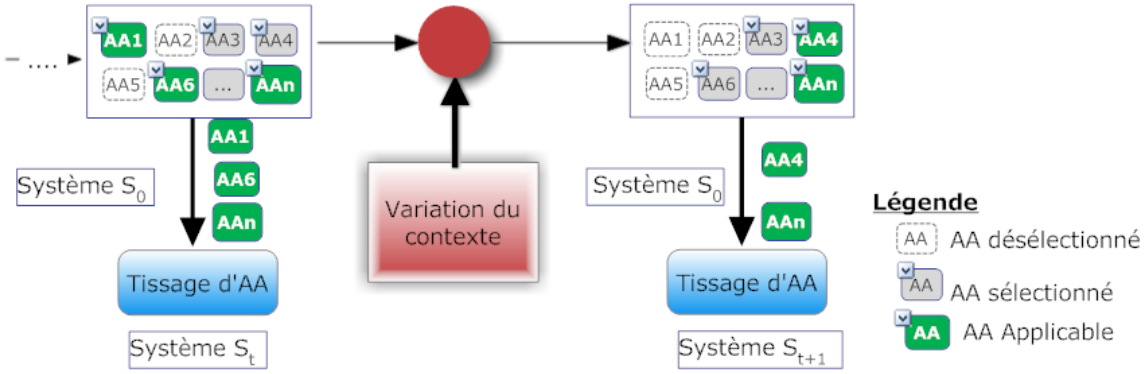


FIGURE 5.6 – Les états des AA est fonction des variations du contexte

Les AA qui ont l'état applicable seront tissés. Le tissage des AA est donc réalisé de manière opportuniste (c'est-à-dire qu'un AA doit être appliqué lorsque l'infrastructure logicielle le permet). Ils sont combinés en fonction de l'application sur laquelle ils s'appliquent, respectant la dynamique d'évolution du contexte d'exécution. La figure 5.6 illustre un exemple où les AA passent par différents états suite à une variation du contexte. L'aspect d'assemblage AA_1 passe de l'état appliqué vers désélectionné (en passant par l'état sélectionné). Cela signifie qu'on se trouve dans une situation où l'aspect ne fait plus partie de la logique d'adaptation nécessaire à l'application. D'autres changements d'états sont possibles ; les AA qui n'étaient pas applicables (AA_4 par exemple) peuvent le devenir si une nouvelle entité apparaît dans le graphe de l'application (de même, ceux qui étaient tissés peuvent redevenir inapplicables, AA_6).

La définition d'un cycle de vie pour les aspects d'assemblage a comme avantage d'offrir la possibilité d'ajuster automatiquement le comportement visé par une adaptation selon le contexte. Par exemple, un AA pourra être appliqué dans une situation (par exemple, à la maison) et n'avoir aucune signification en dehors de cette situation (au travail par exemple). Par conséquent, les AA s'intègrent facilement dans le mécanisme d'auto-adaptation des applications.

Dans la suite nous détaillons notre contribution en utilisant les AAs comme un modèle de spécification des adaptations. Nous verrons que les AAs pourront être spécifiés en se basant sur un langage ou directement à l'aide du formalisme de graphe que nous proposons.

5.2.2 Approches de production des AAs

Les aspects d'assemblage sont conçus pour être tissés sur des applications qui ne sont pas forcément connues pendant leur spécification. Toutefois, nous avons besoin d'une connaissance partielle des applications à adapter c'est-à-dire les éventuelles entités logicielles participant à la composition sans violer la propriété de boîte noire de ces entités. Comme un aspect, un aspect d'assemblage est formé de deux parties : *point de coupe* et *greffon*. La partie de point de coupe a comme objectif de sélectionner les ports des entités logicielles qui font l'objet d'une composition. Ils permettent alors d'identifier les points de jonction sur lesquels les greffons vont être tissés. Dans le modèle AA, un point de jonction est toujours un point du flot d'événements (opération, événement), mais pas de la même granularité qu'en AOP. Un point de jonction est un port d'un composant logiciel. Un greffon décrit la composition effective. Un AA est tissé dans une application en fabriquant son *instance de greffon* avec les mêmes entités que celles de l'application. Il est traduit en un assemblage de composants (graphe) dont certains composants représentent les points de coupe. Le tissage d'un AA produit alors un ou plusieurs assemblages de composants logiciels.

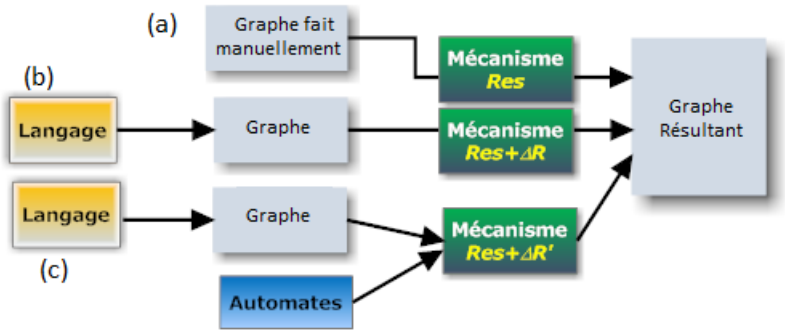


FIGURE 5.7 – Approches pour la spécification des adaptations

La spécification d'un AA pourra être effectuée en utilisant : (1) un langage ou (2) un modèle de graphe. Cela est illustré par la figure 5.7. L'utilisation d'un langage pour décrire les AAs nécessite une transformation des instances du greffon vers le modèle de graphe accepté par le mécanisme de composition.

5.2.2.1 Approches utilisant les langages

Langage de définition des points de coupe Un point de coupe d'un AA vise à sélectionner les endroits dans le graphe de l'application où les instances du greffon seront tissées. La partie de point de coupe d'un AA est une liste de règles dont chacune est spécifiée comme un filtre appliqué sur la liste de points de jonction de l'application. Dans notre implémentation, nous utilisons un langage de filtrage par motif (*pattern matching*) pour identifier les composants ainsi que leur port qui sont identifiés par leur nom. Chacune des règles formant le point de coupe peut sélectionner plusieurs endroits dans le graphe de l'application (à une règle correspond 1 ou plusieurs points de jonction). De ce fait, la partie point de coupe peut produire une liste de points de jonction et par conséquent, le greffon de cet aspect sera instancié plusieurs fois en utilisant toute les combinaisons possibles de ces points de jonction. Un AA n'est tissé que dans le cas où toutes les règles de sa partie point de coupe ont sélectionné au moins un point de jonction.

Langage de définition des greffons Un greffon décrit un ensemble de règles portant sur les points de jonction (les ports de composants). Ces règles définissent un ensemble de composants et de liaisons qui doivent être tissées dans un assemblage de composant. Nous avons vu qu'un greffon définit trois types de règles. Nous présentons ces règles dans le langage ISL4WComp [CFW09] :

- *Instanciation d'un composant*. La règle est sous la forme *comp :type*. L'instantiation d'un composant boîte blanche (les connecteurs) est implicite, c'est à dire qu'il n'y a pas une règle à écrire pour les instancier mais ils seront instanciés dès leur utilisation dans les interactions.
- *Ajouter une interaction*. La syntaxe est $port\_requis \rightarrow (\ port\_fourni\ )$, ou $port\_requis \rightarrow Connecteur(..., port\_fourni)$. Le membre gauche de ces règles est un port de sortie d'un composant (c'est un évènement $\wedge.port$)
- *Réécriture des liens existants*. Cette règle a la même force que la précédente mais la particularité dans ce cas est l'utilisation dans la partie membre droit de la règle un connecteur de type *utilitaire* (c'est à dire $call$, $delegate$, etc.). De cette manière nous exprimons explicitement que notre objectif est de réécrire un lien existant dans l'assemblage de composants.

Exemple d'AA Nous choisissons un exemple qui illustre ces trois types de règles. Cet exemple définit une adaptation pour une application de domotique, liant un interrupteur, un store et un capteur de luminosité. Ces trois dispositifs sont représentés par trois composants dans l'application existante et leurs ports seront identifiés par la partie point de coupe de notre aspect d'assemblage. Nous supposons que l'application de base contient un composant *switch* lié à un composant *Light* (à travers leur ports). La figure 5.8 présente l'AA spécifiant la logique de coordination entre les trois dispositifs.

La partie point de coupe montre que cet AA ne s'applique que dans le cas où les trois dispositifs (switch, store et sensor) sont présents dans l'environnement. Si c'est le cas, la partie greffon est appliquée (ici le greffon est appelé *gestion_Lum*). Les trois paramètres Switch, Store, Sensor représentent les composants fournis par le calcul des points de coupe. Ils seront remplacées par les points de jonction instanciables lors du tissage. La première règle (ligne 7) permet d'instancier le composant *Threshold* qui sera utilisé pour déterminer si le seuil de luminosité est atteint ou non (*threshold.IsReached*). La ligne suivante montre un exemple de réécriture d'un lien existant. Le lien qui sera réécrit est celui qui existe entre Switch et Light. Cette interaction sera conditionnée par un seuil. Pour cela il y a une instanciation implicite d'un connecteur conditionnel IF. Si le seuil est vérifié, alors l'évènement du switch *Êvented_New_Value* va déclencher l'ouverture des stores (*Strore.MoveUp*). Sinon l'interaction existante reste telle qu'elle (c'est à dire cet évènement allume Light). La ligne 10 présente un exemple

```
1  Pointcut:
       Switch:=/Switch[[:digit:]].^ Evented_New_Value/
       Store:=/Store*.MoveUp/
4      Sensor:=/*.^brightness_New_Value*/
   Advice:
       schema gestion_Lum (Switch,Store,Sensor) :
7          threshold: BasicBeans.Threshold(threshold=8)
           Switch −> if(threshold.IsReached){store}
                  else{call}
10     Sensor−> threshold.set_Value
```

FIGURE 5.8 – Un aspect d'assemblage pour la gestion de la lumière

de création d'une interaction simple de type port à port. Si la valeur de la luminosité externe change alors une notification est envoyée au composant threshold pour qu'il mette à jour sa valeur.

La spécification des AAs utilise principalement un ensemble de connecteurs de base et des opérateurs liés au langage défini. Cependant, un AA pourra utiliser une nouvelle logique de composition en définissant un nouveau connecteur. Dans ce cas, le comportement de ce dernier est fourni sous la forme d'un automate (qui peut être sous forme de langage ou graphique). Ceci est montré par la partie *(c)* de la figure 5.7.

5.2.2.2 Approche utilisant les graphes

Une autre manière de spécifier des AAs est de les décrire directement en utilisant le formalisme de graphe [FLT+12]. La figure 5.9 présente le même exemple que l'aspect d'assemblage de la figure 5.8 mais cette fois exprimé à l'aide du modèle du graphe. La différence majeure entre ces deux représentations réside dans l'ensemble de connecteurs qui sont autorisés à être utilisés. Nous avons évoqué précédemment (section 3.3.2) que les langages emploient des opérateurs spéciaux pour piloter le mécanisme de résolution d'une manière externe. Cependant, le modèle de graphe n'inclut que les connecteurs de bases (dans l'exemple il n'y a pas l'opérateur *CALL*).

Quelque soit le mécanisme utilisé dans la définition des AA, le processus de leur tissage reste le même (ce qui diffère c'est l'implémentation de la fonction d'instanciation des greffons qui inclut une phase de transformation vers les graphes). La description du processus de tissage fera l'objet de la section suivante.

5.3 Tisseur d'aspects d'assemblage

Le tisseur est le programme responsable de l'application des aspects pour construire systématiquement le graphe unique résultat représentant l'application finale. Pour se faire, il calcule l'ensemble des modifications structurelles à partir du graphe de base de l'application et des graphes instanciés à partir des greffons des aspects d'assemblage. L'architecture du tisseur est illustrée par la figure 5.10. Elle est décomposée en deux parties. La première traite des AAs et a pour but de transformer les greffons de ces derniers en graphes. La seconde partie du tisseur est la *composition* de ces graphes. Elle travaille uniquement avec les graphes qui sont les instances de greffon générées par la fabrique ainsi

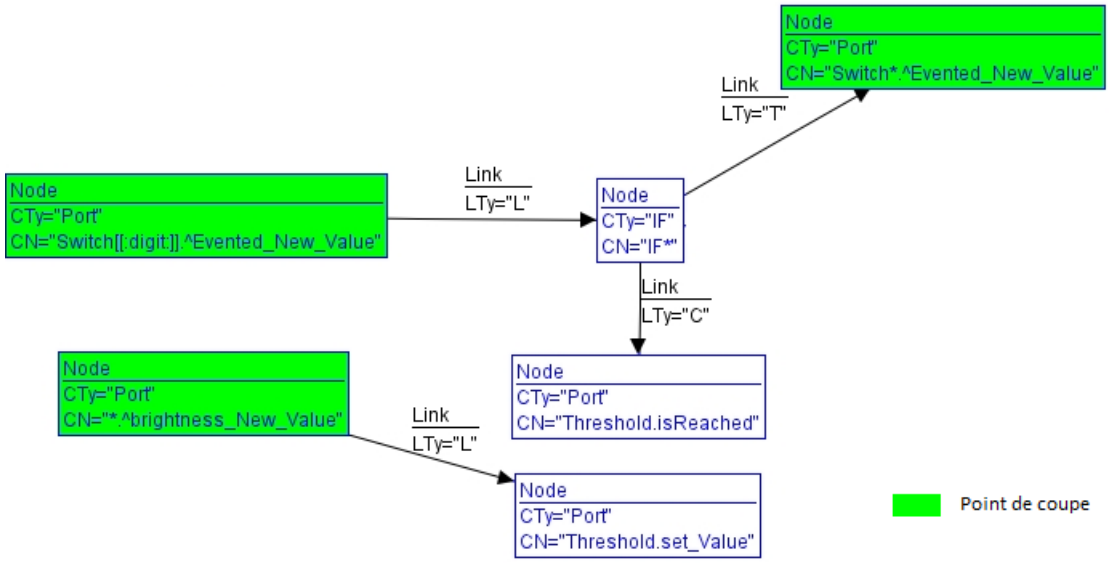


FIGURE 5.9 – Une adaptation exprimée directement à l'aide du formalise de graphe

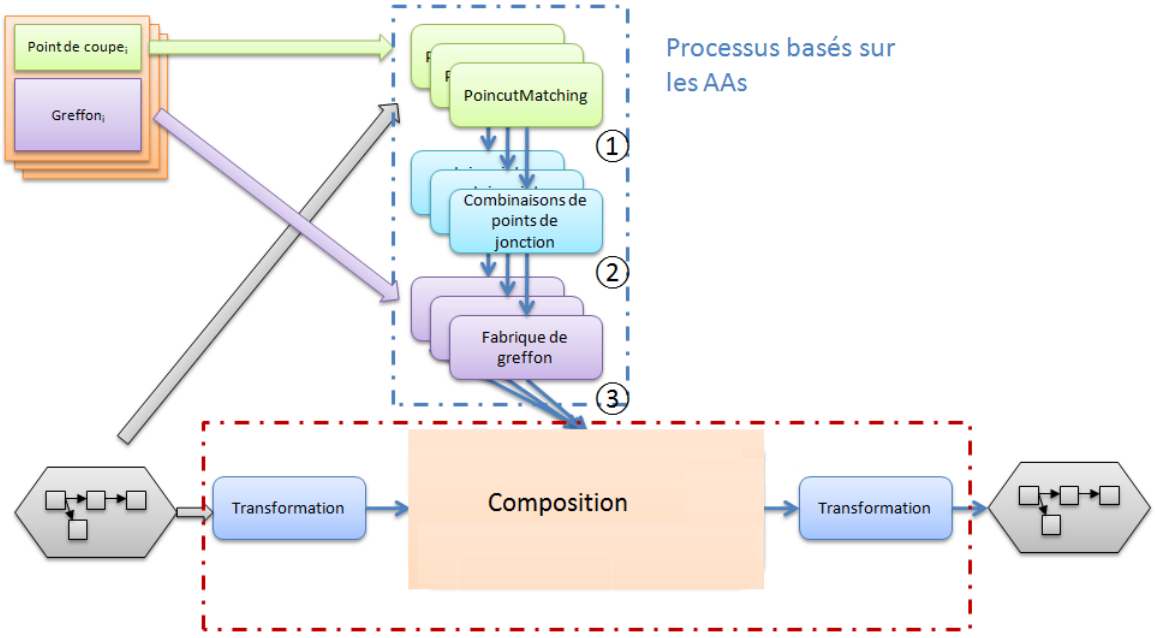


FIGURE 5.10 – Le tisseur des Aspects d'Assemblage

que le graphe de l'application de base. Une description détaillée ainsi qu'une formalisation du tisseur sont fournies dans [Fer11]. Nous présentons brièvement l'opération d'instantiation des AAs et nous nous focalisons sur l'opération de transformation des instances de greffon vers des graphes.

5.3.1 Instanciation des greffon des AAs

Les aspects d'assemblage qui ont été sélectionnés sont évalués pour déterminer s'ils seront appliqués. Cette évaluation (*Pointcut Matching*, étape (1) dans 5.10) identifie l'ensemble des points de

jonction sur lesquels peut s'appliquer l'AA. Nous avons vu que lorsque des points de jonction ont été identifiés pour chacun des points de coupe d'un AA, il devient applicable. Les points de jonction vont remplacer les composants variables représentant les points de coupe dans l'expression des greffons. De cette manière des instances de greffons sont créés (fabrique de greffon, étape (3) dans 5.10) correspondant à la mise en oeuvre des adaptations génériques dans une application réelle en utilisant les composants présents dans l'infrastructure.

5.3.2 Transformation : Instance de greffon $\leftrightarrow$ un graphe

Une instance de greffon est un ensemble de règles qui constituent des opérations à effectuer au niveau de l'application de base. Ces opérations sont : *ajouter un noeud* ($add\_Node(CTY, CN)$) et *ajouter un lien* ($add\_link(N\_source, N\_dest, Etiquette)$). Nous avons vu précédemment que la suppression des nœuds n'est possible que suite à la disparition d'un dispositif et donc lorsqu'un AA est détissé. Ce choix est nécessaire pour avoir la propriété de symétrie. Nous présentons dans le tableau 5.2 l'ensemble des opérations élémentaires au niveau du graphe qui correspond à l'AA de l'exemple précèdent la figure 5.8.

TABLE 5.2 – Correspondance entre les instruction du langage et les actions au niveau du graphe

Règles	Graphe : Opération élémentaires
threshold1 : BasicBeans.Threshold	Ajouter avec add_node() un nœud pour chaque port de ce composant
Switch -> if(threshold1.IsReached){store}	$add\_node(IF, IF1)$
	$add\_link(Switch1.Evented\_New\_Value, IF1, "L")$
	$add\_link(IF1, threshold1.IsReached, "C")$
	$add\_link(IF1, Store1.MoveUp, "T")$
else{call}	$add\_node(CALL, call1)$
	$add\_link(IF1, call1, "F")$
Sensor-> threshold1.set_Value	add_link(Sensor1.brightness_New_Value ,threshold1.set_Value ,"L")

La figure 5.11 expose le graphe résultat de la transformation de l'instantiation de l'AA de notre exemple.

5.3.3 Extension du tisseur

Tous les concepts qui ont été présentés ont été implémentés dans la plate-forme expérimentale WComp. L'architecture de WComp s'organise autour de containers et de designers. L'objectif des containers est de prendre en charge dynamiquement la gestion de la structure tels que l'instanciation, la destruction des composants et des liaisons. Une applications est créée par un assemblage de composants, dans un container WComp, conformément au modèle de composants légers LCA. WComp nous permet de mettre en œuvre une application à partir d'une orchestration des services disponibles dans l'environnement et d'autres composants sur étagères. Le tisseur est à son tour implémenté comme un assemblage de composants encapsulé dans un autre container. Le tisseur va agir sur le container de l'application pour l'adapter. Nous avons apporté des modifications à l'assemblage implémentant le tisseur d'Aspects d'Assemblage par l'ajout de trois composants pour réaliser la résolution des interférences à l'aide des transformations de graphe. Ceci est illustré par la figure 5.12.

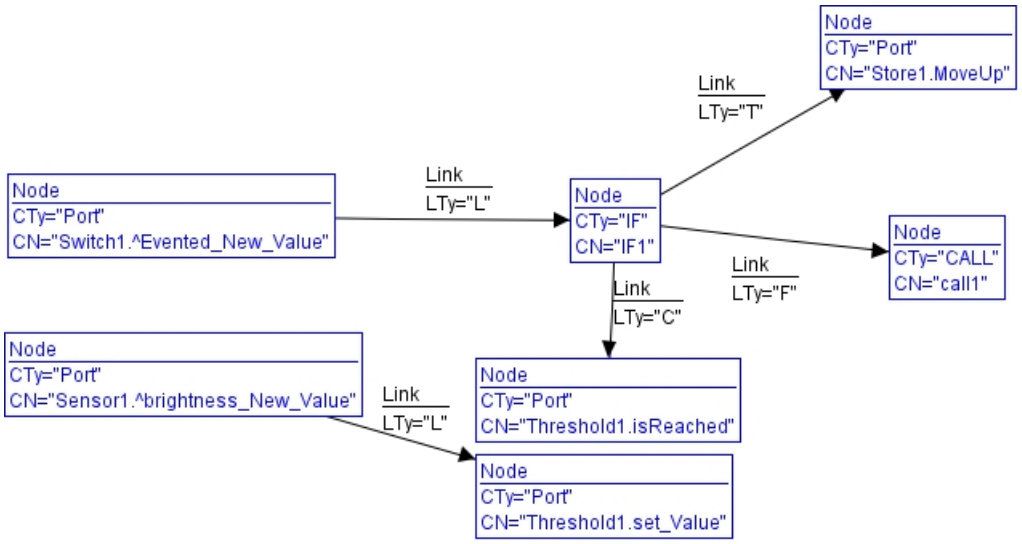


FIGURE 5.11 – Résultat de transformation d'instance d'AA vers un graphe

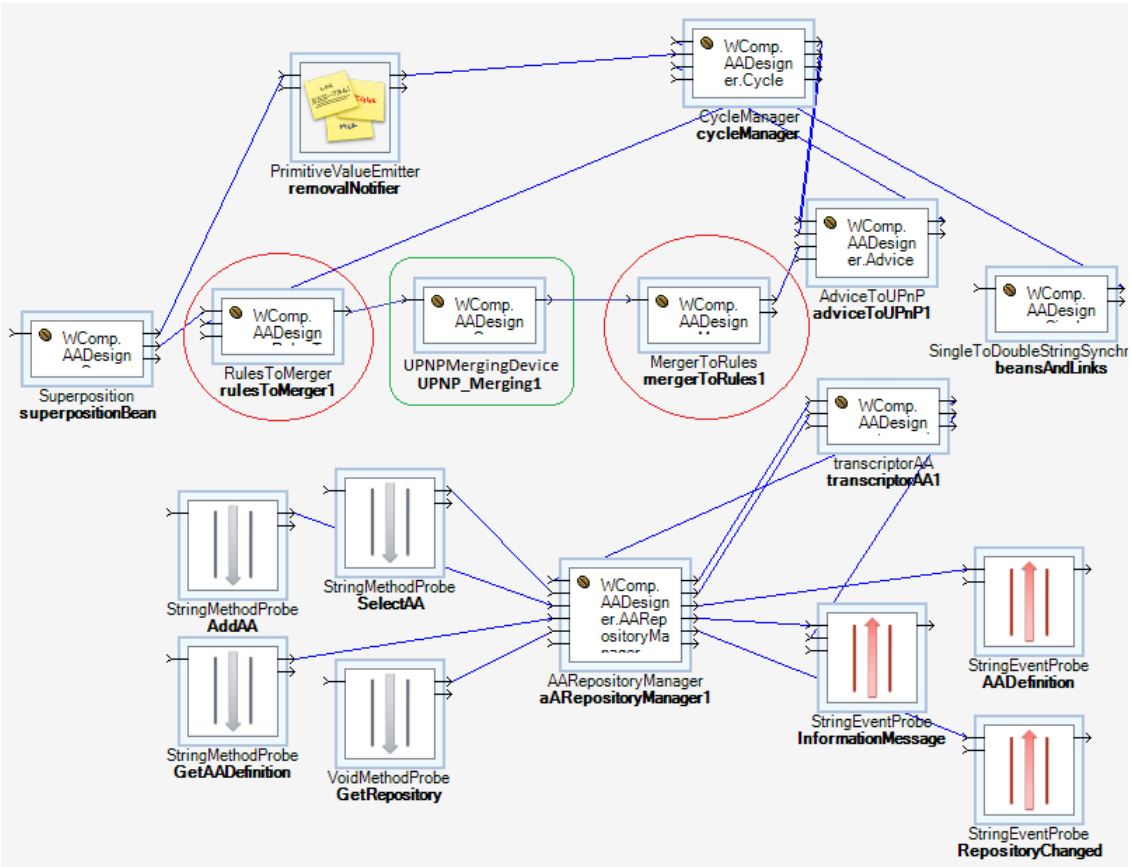


FIGURE 5.12 – L'assemblage de composants définissant le tisseur (encapsulé dans un container)

Les deux composants mis en évidence par un cercle rouge sont responsables de la transformation depuis la plate-forme vers notre modèle de graphe (*RulesToMerger*) et depuis notre modèle de

graphe vers la plate-forme (MergerToRules). Le processus de gestion des interférences est assuré par le composant *UPNPMerginDevice*. Le moteur de transformation de graphe que nous utilisons est implémenté à l'aide de AGG, lui-même implémenté en Java. La communication entre notre plate-forme et notre moteur de fusion est faite à l'aide d'UPnP. Pour cela notre moteur de fusion a été encapsulé dans un service UPnP et ajouté comme un composant proxy dans le tisseur [FBBLT+11].

Jusqu'à présent, nous avons défini tous les éléments nécessaires pour le déroulement de notre approche de composition. Dans la section suivante, nous allons étudier un exemple concret de mise en oeuvre de nos règles d'adaptation et de leur composition.

5.4 Expérimentation : application dans le domaine domotique

5.4.1 Scenarios

Marie est une personne âgée qui rencontre depuis quelque temps des difficultés dans la gestion de son domicile. Pour cela, elle a contacté plusieurs fournisseurs de solution d'assistance. Elle a choisi dans divers catalogues, les comportements qu'elle souhaite avoir chez elle. Nous présentons les comportements qui provoquent des problèmes d'interférences :

- *Informations pratiques : Un ensemble d'informations (comme la météo, les transports en commun, etc.) seront diffusées sur l'un des dispositif qui a la capacité d'affichage*
- *Santé*
  - *Un message sera affiché à l'heure de la prise du médicament en utilisant un dispositif dans la pièce dans laquelle la personne se trouve.*
  - *Une alerte sera lancée lorsqu'il pleut et que les fenêtres sont ouvertes.*
  - *S'il y a une personne dans une pièce et que la température dépasse un seuil malgré le fait que les volets et fenêtres sont ouverts, alors il faut : (1) fermer les fenêtres et (2) faire fonctionner la climatisation.*
- *Économie d'énergie*
  - *Lorsqu'il y a un dispositif qui mesure la luminosité externe, alors les volets s'ouvrent lorsque il y a suffisamment de luminosité externe à la place d'allumer des lampes.*
  - *Une alerte sera lancée si le chauffage est mis en marche alors que les fenêtres sont ouvertes*
- *Automatisation des tâches : Le système d'arrosage du jardin se lance automatiquement selon une fréquence choisie par Marie avec la possibilité de le commander aussi à distance par un dispositif (téléphone, tablette, etc.).*

Ce scénario nous permet d'illustrer certaines contraintes imposées par le cadre d'informatique ambiante (1.1.2) :

Diversité d'entités Ces spécifications de comportements mettent en œuvre des logiques de composition entre diverses entités. Le nombre d'entités concernées n'est pas fixé à l'avance. Il y aura toujours la possibilité d'ajouter de nouveaux dispositifs.

Imprévisibilité L'imprévisibilité est liée à la disponibilité des dispositifs dans la maison. Cette disponibilité est n'est pas connue à priori. Les comportements seront assurés en utilisant les capacités offertes par les dispositifs qui sont disponibles en remplaçant ceux qui deviennent indisponibles.

Par exemple le premier comportement sera mis en œuvre en utilisant un dispositif qui a la capacité d'affichage. Il pourra être mis en œuvre suivant les dispositifs disponibles comme une télévision, un téléphone ou une tablette.

Extensibilité La liste des comportements intégrés dans la maison de Marie peut évoluer à tout moment. Il est possible d'intégrer des nouveaux comportements suite à l'apparition de nouveaux besoins. Nous illustrons l'extensibilité par l'ajout d'un nouveau comportement qui va permettre d'ajuster le système d'arrosage. L'extensibilité touche aussi la spécification de ces comportements en ajoutant des nouvelles logiques de coordinations (par l'ajout de nouveaux connecteurs).

Indépendance des adaptations Les comportements choisis par *Marie* sont spécifiés par des fournisseurs différents. Ces compositions sont donc spécifiées indépendamment les unes des autres et elles n'ont pas été conçues pour fonctionner ensemble.

5.4.2 Description des AAs

Les comportements choisis par *Marie* sont définis sous formes d'Aspects d'Assemblage. La figure 5.13 présente l'aspect d'assemblage responsable de l'affichage des informations utiles pour *Marie*. Pour s'appliquer, cet aspect à besoin d'un service dans l'environnement qui envoi l'évènement $info$ et d'un dispositif d'affichage. La partie greffon met en relation ces deux entités : le port de sortie $info$ est lié au port *display* ce qui permettra d'afficher les données portées par cet évènement de sortie. La spécification de cet AA ne se limite pas à un service particulier. Il sera appliqué à tous les services qui satisfont la règle de la partie point de coupe. À ce niveau nous n'avons aucune connaissance de la nature des informations qui pourront être affichées. La figure 5.14 illustre l'AA permettant d'afficher un message à l'heur de prise des médicaments.

```
Pointcut:
    info:=/Serv[[:digit:]]*.^info/
    display:=/*.Display/

Advice:
    schema gestion_info (info,display) :
        info -> display
```

FIGURE 5.13 – Un aspect d'assemblage pour la gestion des informations

```
Pointcut:
    display:=/*.Display/

Advice:
    schema gestion_medicament (display) :
  TimerMed: BasicBeans.Timer(Interval:12, Msg:"Med")
    Timer.^TimeOut -> display
```

FIGURE 5.14 – Un aspect d'assemblage pour le rappel de la prise de médicament

Un autre comportement a été choisi pour régler la température de la maison. L'état de santé de *Marie* fait qu'elle est sensible à la chaleur. Même si elle n'y prête pas attention, il faut que la température de la maison soit gérée automatiquement pour la faire baisser rapidement. L'AA de la figure 5.15 illustre ce comportement. Cet exemple montre qu'il est possible d'utiliser les deux classes de connecteurs dans la définition d'un AA. La ligne 10 présente la logique de composition entre les différentes entités impliquées dans la mise en œuvre de ce comportement. S'il y a une personne dans la pièce et que la fenêtre est déjà ouverte et si le seuil de la température a été dépassé, alors dans ce cas la fenêtre sera fermée et la climatisation sera mis en marche.

```
Pointcut:
    temp:=/temperature∗.^Evented_NewValue/
    Window:=/Window∗.MoveDown/
  Clim:=/Clim∗.setStatus/
    WindowState:=/Window∗.^isUp/
    Sensor:=/∗.^Sombody∗/
Advice:
    schema gestion_Lum (temp, Window, WindowState, Clim, Sensor) :
        threshold: BasicBeans.Threshold(threshold=8)
        AND(sensor, WindowState) −> if(threshold.IsReached){par(Window, Clim)}
        Sensor−> threshold.set_Value
```

FIGURE 5.15 – Un aspect d'assemblage pour la gestion de la climatisation

La figures 5.16 présente le comportement relatif au déclenchement d'une alerte si le chauffage est mis en marche alors qu'il existe une fenêtre ouverte. La figure 5.17 montre l'aspect d'assemblage qui va émettre une alerte s'il pleut et que la fenêtre est ouverte. Nous verrons plus tard qu'une interférence aura lieu entre ces deux comportements (il s'agit de deux préoccupations différentes et indépendantes).

```
Pointcut:
    Heater:=/Hieater[[:digit:]]∗.^isOn/
    WindowState:=/Window∗.^isUp/
    EnergyWarning= Alert.setOn
Advice:
    schema gestion_Energy (Heater,WindowState,EnergyWarning) :
      AND(Heater, WindowState) −> EnergyWarning
```

FIGURE 5.16 – Un aspect d'assemblage pour la gestion de l'énergie

```
Pointcut:
    Sensor:=/Sensor[[:digit:]]∗.^Raining/
    WindowState:=/Window∗.^isUp/
    SecurityWarning= Alert.setOn

Advice:
    schema Health_Warning (Sensor,WindowState,EnergyWarning) :
     AND(Sensor, WindowState) −> EnergyWarning
```

FIGURE 5.17 – Un aspect d'assemblage pour la santé

Pour ne pas oublier l'arrosage de son jardin, Marie a choisi un comportement permettant de commander cette tâche automatiquement. L'arrosage se déclenche tous les deux jours ou bien explicitement par Marie en utilisant son téléphone (figure 5.18).

5.4.3 Application du scénario

Nous avons vu que les Aspects d'Assemblages s'appliquent en fonction des dispositifs présents dans l'environnement. Si la partie point de coupe de chacun de ces AA est satisfaite, alors une ou

```
1  Pointcut:
       DispInt:=/Phone∗.^StartArro/
       arrosage:=/arrosage∗.Start/
4
   Advice:
       schema gestion_arrosage (DispoInte,arrosage) :
7    TimerMed: BasicBeans.Timer(Interval:48)

       OR(DispoInte, Timer.^TimeOut) −> arrosage
```

FIGURE 5.18 – Un aspect d'assemblage pour l'arrosage du jardin

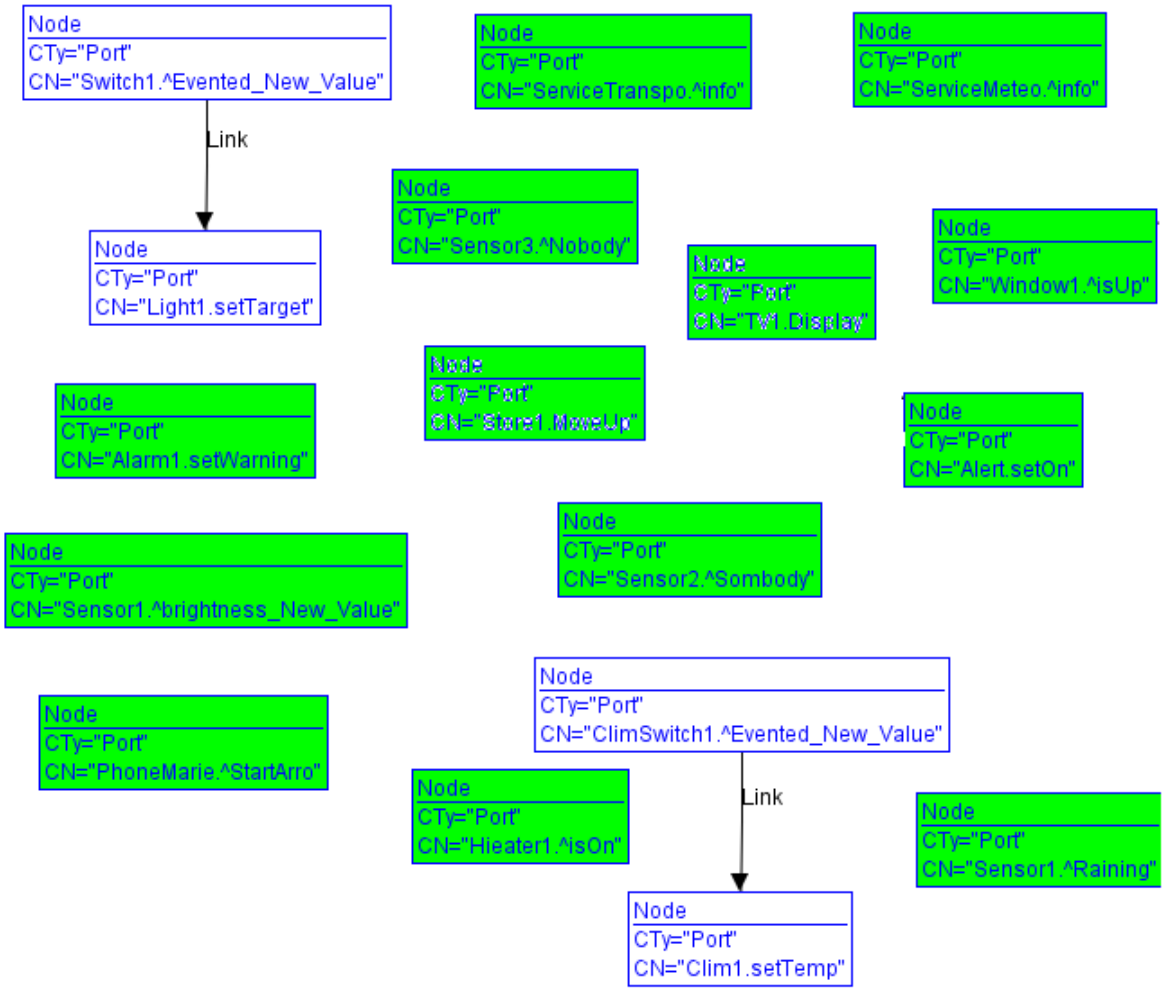


FIGURE 5.19 – Le graphe de l'application de base avant le processus d'adaptation

plusieurs instances du greffon peuvent être ajoutées au graphe de l'application initiale. Donc l'ensemble des adaptations à appliquer est variable et elles seront composées de manière non-anticipée. Nous supposons que le câblage existant dans la maison est celui illustré dans la figure 5.19. C'est

l'application de base sur laquelle nos AAs seront tissés. Les nœuds en vert représentent les ports des dispositifs ajoutés dans la maison pour mettre en œuvre les nouveaux comportements souhaités.

5.4.3.1 Le tissage des AAs et la génération des sous-graphes

Le tissage du l'AA de la figure 5.13 donne naissance aux deux graphes 5.20 et 5.21 (ceci est du au faite qu'il y a deux services disponibles, météo et transport, et qui permettent d'envoyer des informations). Le graphe présenté par la figure 5.22 est le résultat de l'instantiation du greffon de l'AA de la figure 5.14. L'instance du greffon de l'AA de la gestion de la température (figure 5.15) est illustré dans la figure 5.24. D'autres instances d'AAs sont présentées par la figure 5.23.

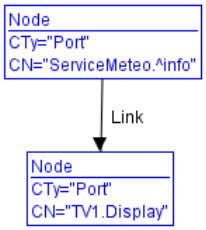


FIGURE 5.20 – La première instance pour l'AA 5.13

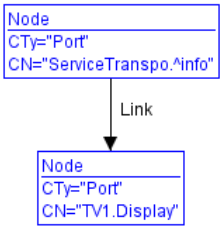


FIGURE 5.21 – La deuxième instance pour l'AA 5.13

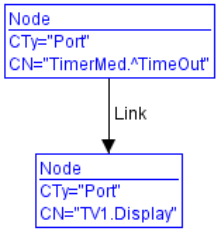


FIGURE 5.22 – Instance de l'AA 5.14

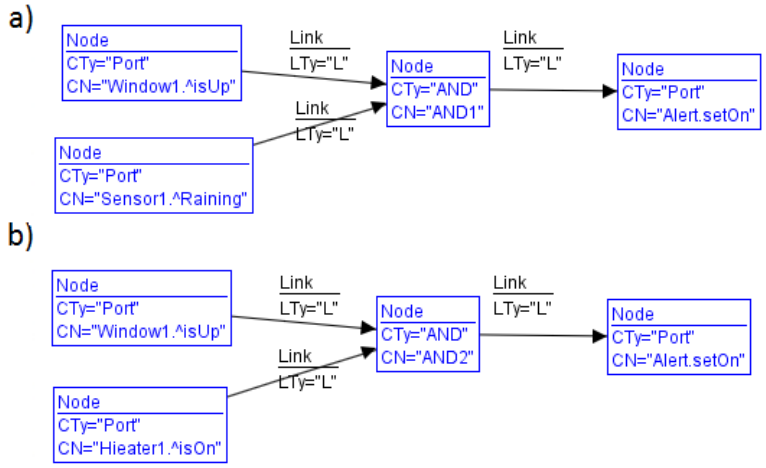


FIGURE 5.23 – Deux instances pour l'AA la figure 5.23

Dans la suite nous déroulons le mécanisme de composition en utilisant ces instances d'Aspects d'Assemblage.

5.4.3.2 La Composition

La superposition Les comportements déjà ajoutés par *Marie* vont être regroupés ensemble pour former le système final. La première étape de la composition consiste à superposer tous les sous-graphes obtenus suite à l'instantiation de ces différents Aspects d'Assemblage (c'est l'union des graphes des figures 5.19, 5.20, 5.21, 5.22, 5.24, 5.20 et 5.11).

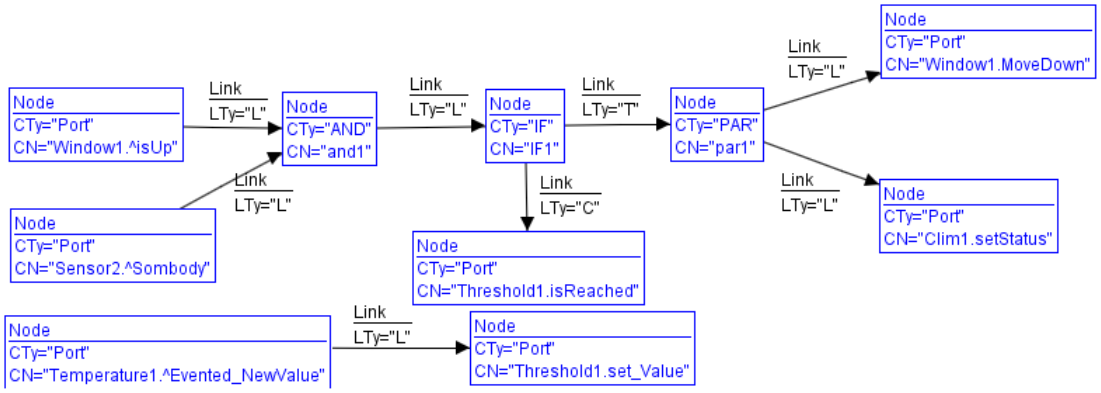


FIGURE 5.24 – Instande de l'AA de la figure 5.15

La Détection des interférences Le graphe résultant de l'étape de la superposition présente dans notre cas des problèmes d'interférences. La figure 5.25 illustre l'ajout de nœud *Fus* (⊗) pour marquer les endroits où il y a interférence. Un premier nœud de fusion marqué une interférence de type 1-N

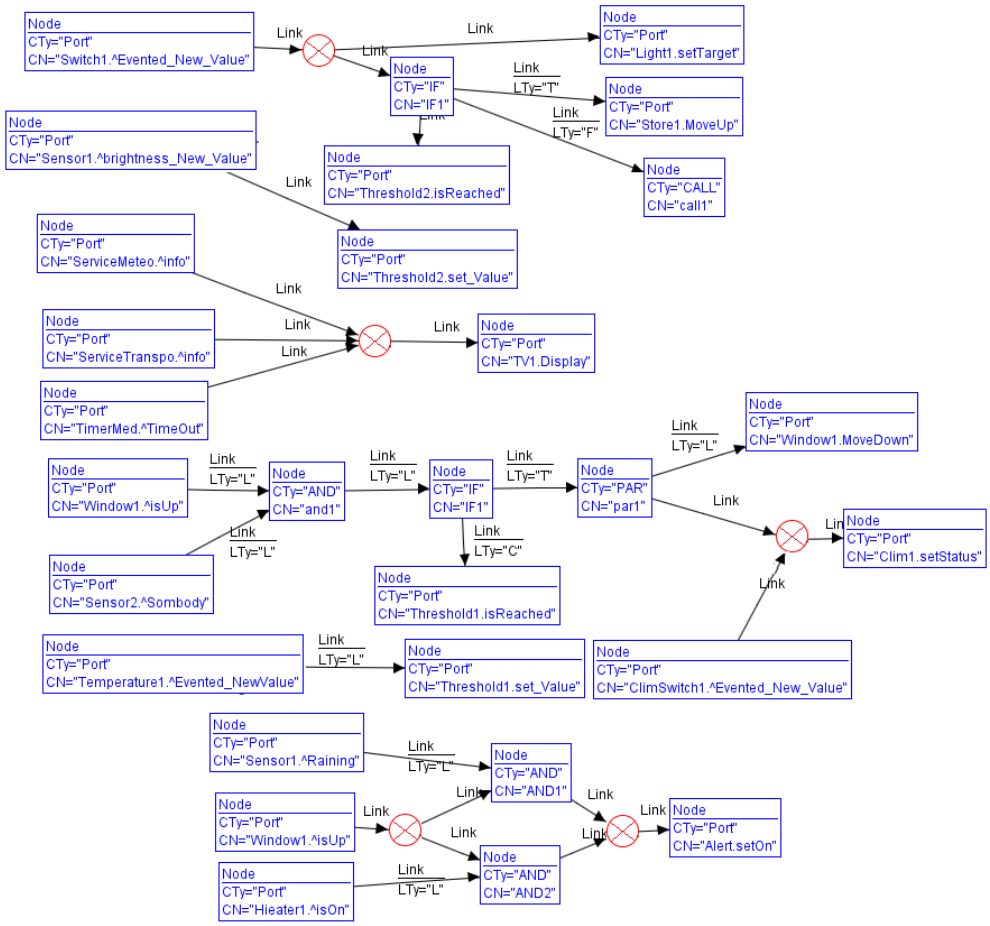


FIGURE 5.25 – L'ajout des marqueurs ⊗ pour les problèmes d'interférences

au niveau du port de sortie *Evented_New_Value* du composant *Switch*1. Selon cette configuration, la

lampe sera allumée quelle que soit la valeur de la luminosité externe. On ignore alors le comportement souhaité pour faire des économies d'énergie. Le deuxième nœud $\otimes$ a été ajouté à l'entrée du port *Display* de la télé. C'est un problème d'accès concurrent (N-1). Le message qui sera affiché sur l'écran de la Télé dépend de l'ordre dans lequel le port a été appelé par les différentes entités en question. S'il y a une occurrence de trois évènements de sortie (*ServiceTranspo.*$\wedge$ *info, ServiceMeteo.*$\wedge$ *info et TimerMed.*$\wedge$ *info*), alors seul le dernier message qui arrive sera affiché (puisque la réception d'un message écrase l'ancien). Le dernier nœud de fusion marque une interférence de type N-1 à l'entrée du port *SetStatus* du climatiseur.

La résolution des interférences Nous présentons dans ce paragraphe la résolution des interférences détectées dans notre scénario. Ce qu'il faut retenir ici, c'est que la propriété de symétrie permet en effet de ne pas avoir à mettre en place un mécanisme qui permet de faire la distinction entre les Aspects d'Assemblage. Les interférence entre les aspects sont gérées à partir d'un ensemble de règles de transformation de graphes. Ces règles se basent sur la connaissance de la sémantique des connecteurs utilisés dans la spécification des adaptations. La résolution de la première interférence (la partie en haut de la figure 5.25) fait appel à une seule règle de réécriture du graphe qui sera appliquée deux fois. Cette règle c'est la réécriture d'un connecteur *IF* possédant un connecteur *CALL* comme successeur (la règle de la figure A.13 dans annexe A). La partie en haut de la figure 5.26 montre que la solution est de remplacer le connecteur *CALL* de la branche *False* du connecteur *IF* par le lien qui a causé l'interférence. De ce faite, la lampe ne sera allumée que dans le cas où il n y a pas suffisamment de luminosité externe (sinon ce sont les stores qui vont être ouverts).

La résolution de deux autres interférences fait appel à la même règle de réécriture puisqu'il s'agit de problèmes d'accès concurrents (port $TV1.display$ dans la figure 5.25). Dans ce cas, nous appliquons la règle de résolution par défaut qui consiste à ajouter un ou plusieurs connecteurs *OR*. Dans notre exemple, il y a trois accès concurrents au port Display de la Télé. **Dans ce cas un ensemble de connecteurs synchrones seront ajoutés à l'entrée de ce port pour assurer la synchronisation**. La figure 5.26 illustre ces connecteurs qui devront être synchronisés (ajouter un préfixe commun *CComplex* à ces connecteurs). Ce préfixe servira lors de la phase de transformation du graphe vers la plate-forme pour garantir que cette partie de l'assemblage de composants est synchrone.

Nous avons déroulé dans cette première partie de notre scénario le processus de composition des adaptations qui embarque une phase de gestion des interférences. Notre contribution principale est de pouvoir étendre dynamiquement et automatiquement ce mécanisme de gestion des interférences pour supporter la résolution des cas d'interférences qui n'ont pas été conçus pour les traiter. Dans le paragraphe suivant, nous ajoutons une nouvelle adaptation dans notre système. La particularité est que cette dernière emploie un nouveau connecteur de coordination qui n'est pas connu par notre mécanisme de gestion des interférences.

5.4.3.3 Extension dynamique du mécanisme de gestion des interférence

La facture d'eau de Marie a augmenté. Ceci est dû au système d'arrosage automatique qu'elle utilise pour son jardin. Marie veut bien soigner son jardin mais elle veut aussi baisser ses factures. Ce nouveau contexte permet de sélectionner une nouvelle adaptation. Cette adaptation va s'intégrer automatiquement dans le système. La nouvelle adaptation a comme objectif d'ajuster le niveau d'arrosage fonction de la sécheresse du terrain. Un capteur multi fonction est déjà installé dans le jardin.

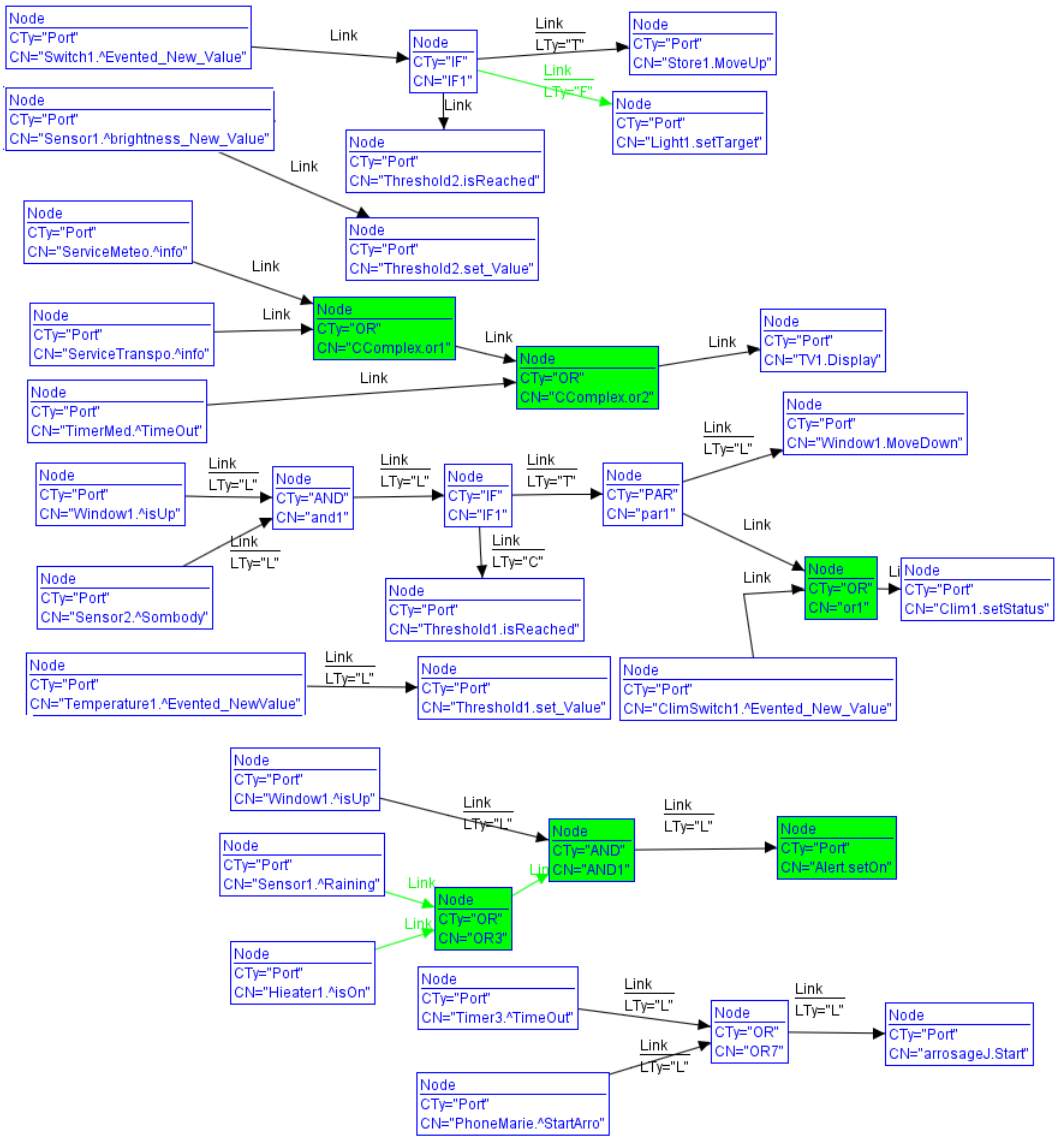


FIGURE 5.26 – Le graphe résultant de l'étape de la résolution des interférences

Il va être utilisé cette fois ci pour mesurer le degré de sécheresse et déterminer par la suite le niveau d'arrosage (très faible, faible, moyen et fort). Ce comportement est décrit par l'AA de la figure 5.27

Le système d'arrosage est réglé par défaut pour être déclenché tous les deux jours. Cette adaptation ajoute (par rapport à l'ancienne adaptation de la figure 5.18) l'utilisation des informations délivrées par le capteur. Le service embarqué dans ce dernier détermine le mode d'arrosage en fonction de l'humidité du terrain. Pour bien comprendre le comportement visé par cette adaptation, nous avons besoin de connaître la sémantique du connecteur *ORDER* qui ne fait pas partie de l'ensemble de connecteurs de base. Notre hypothèse de travail est que cette description d'adaptation a été fournie avec l'automate qui décrit le comportement du connecteur *ORDER*. Cet automate est illustré par la figure 5.28. Ce connecteur est utilisé pour imposer un ordre entre ces deux événements d'entrée. En effet, $out1 = ORDER(a,b)$ signifie que l'appel de port $out1$ s'effectue selon deux modalités. Si

```
Pointcut:
        humid:=/humid*.^Evented_NewValue/
        arrosage:=/arrosage*.start/

Advice:
        schema gestion_arrosage (humid, arrosage) :
                Timerarro: BasicBeans.Timer(Interval:48)
                ORDER(humid, Timerarro) -> arrosage
```

FIGURE 5.27 – Un aspect d'assemblage pour le réglage du mode d'arrosage

l'événement *a* est reçu en premier, la réception par la suite de l'événement *b* permettra d'effectuer un deuxième appel de ce port. Dans le cas où l'événement *b* est activé en premier, il faudra attendre l'activation de *a* pour émettre un appel avec les paramètres porté par *a* et ensuite un deuxième appel avec les paramètres porté par *b*. Dans notre exemple, si le délai de 48h est écoulé, il faudra attendre l'émission des informations du capteur de sécheresse avant de déclencher l'arrosage.

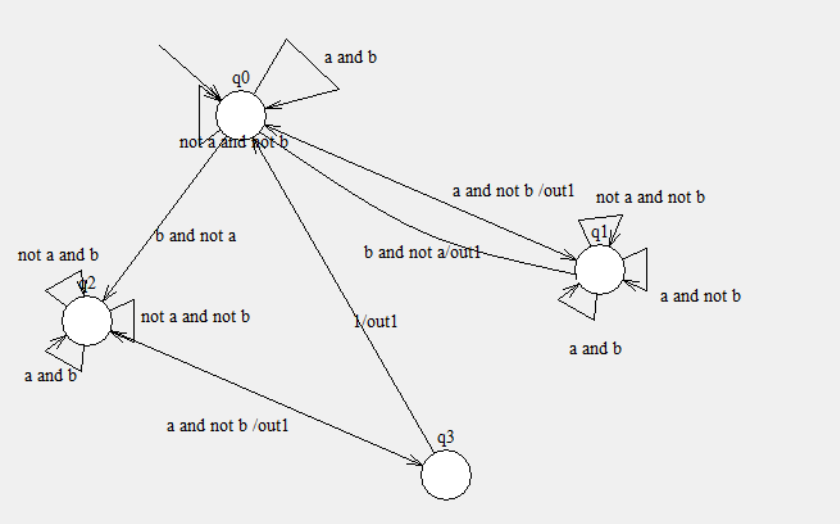


FIGURE 5.28 – L'automate du nouveau connecteur *ORDER*

La composition de cette nouvelle adaptation avec le système donne naissance à un nouveau problèmes d'interférence (accès concurrent). Nous représentons dans la figure 5.29 seulement la partie du système où l'interférence a été détectée. Il s'agit d'une interférence entre les connecteurs *OR* et *ORDER*. Le processus qui a été décrit dans le chapitre 4 (4.2.4.2) va être déroulé pour ce cas concret d'interférence.

La figure 5.30 montre la succession des étapes à faire. Dans notre implémentation, nous avons utilisé un ensemble d'outils qui permettent la manipulation des automates :

Galaxy C'est un éditeur d'automates d'états finis qui a été développé dans un cadre recherche par Daniel Gaffe [Gaf12]. Il regroupe en fait trois logiciels indépendants : un éditeur d'automates simples (mode «*basic automaton*»), un éditeur d'automates hiérarchiques (mode «*LightEsterel*»), un éditeur de *SyncCharts* (mode «*SyncCharts*»). Nous avons utilisé l'éditeur d'automates

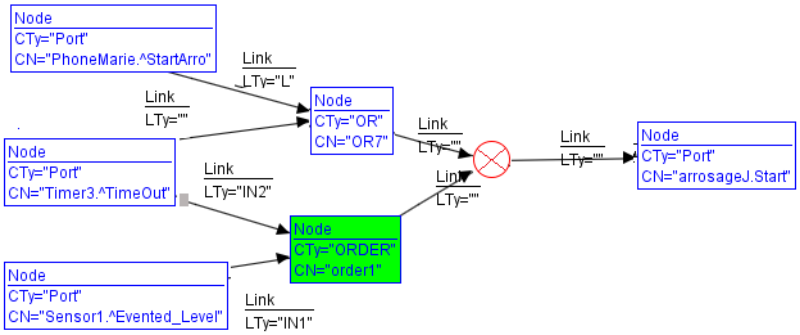


FIGURE 5.29 – Détection d'interférence lors de l'utilisation du nouveau connecteur

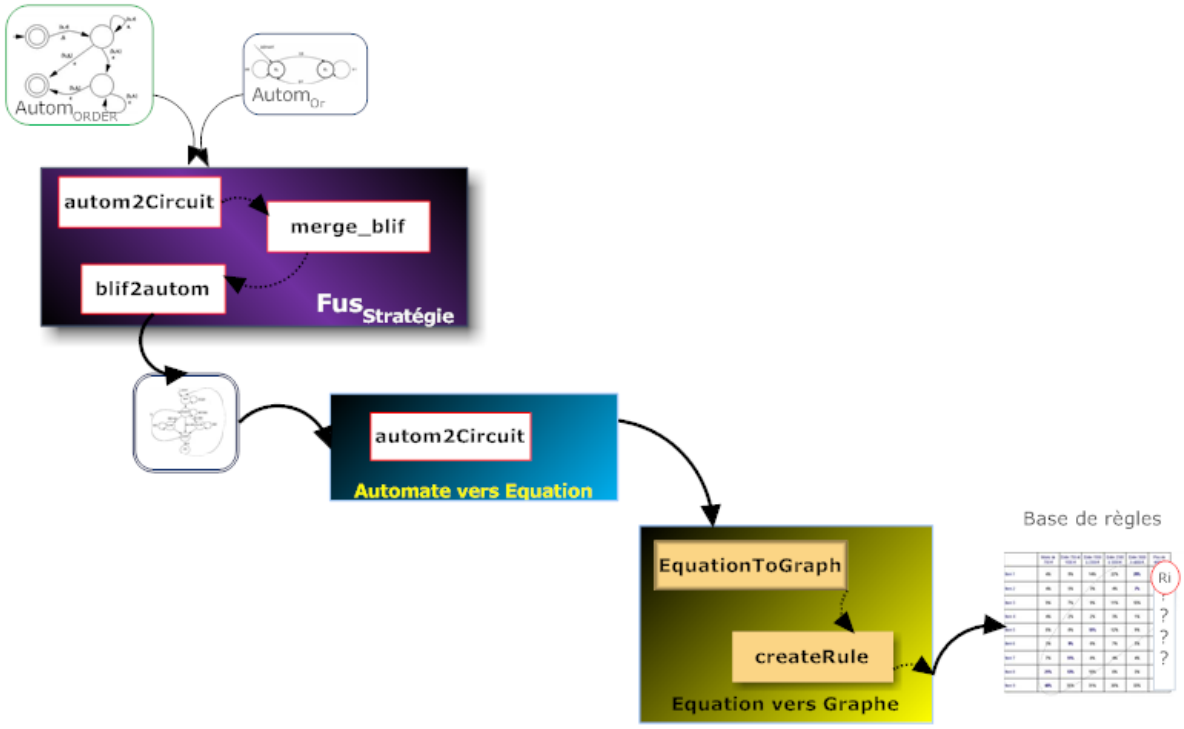


FIGURE 5.30 – Les outils utilisés dans notre implémentation

simples pour spécifier et valider les automates de nos connecteurs.

autom2circuit C'est un synthétiseur d'automates explicites de Galaxy en machine de Mealy. Ce logiciel s'utilise en invité de commandes et sait gérer de multiples formats de sortie suivant les options choisies.

mergeBlif C'est un outil de concaténation de fichiers *blif*. Il effectue la composition des automates (exprimés au format *blif*). Le format *blif* (Berkeley Logic Interchange Format) est une description textuelle d'un automate.

blif2autom Cet outil effectue la conversion d'un fichier *blif* vers un automate accepté par l'éditeur *Galaxy*. *blifcheck* est un vérificateur symbolique de propriétés logiques pour un automate implicite *blif*. Il permet de vérifier si l'automate respecte certaines propriétés. Le processus de

composition a le choix de committer l'adaptation après une phase de test réussie du résultat obtenu ou bien d'annuler l'étape et de revenir à un état précédent.

Nous supposons que tous les automates ont été édités à l'aide de *Galaxy* et validés en utilisant *autom2circuit*. La phase de validation consiste à s'assurer qu'il n'y a pas des comportements non spécifiés par l'automate. La première étape est la composition de ces automates. Pour pouvoir utiliser *mergeblif*, il faudra convertir ces automates vers ce format à l'aide de autom2circuit (avec l'option -blif). L'outil *mergeblif* effectue une composition parallèle d'automates. Le fichier *blif* représentant l'automate résultat de la composition va être transformé de nouveau vers le format automate à l'aide de *blif2autom*. La deuxième étape est le calcul des équations décrivant le comportement de cet automate. Nous utilisons de nouveau autom2circuit pour générer un nouveau fichier au format lustre (avec l'option -ba). Ce fichier contient les équations booléennes décrivant le comportement de l'automate en question. La dernière étape dans notre processus est la génération de la règle de réécriture pour ces connecteurs. Nous avons implémenté les deux fonctions *EquationToGraph* et *createRule* responsables de la création d'une règle conforme au format AGG. Cette règle va être sauvegardée dans la base de règles et sera utilisée pour des cas similaires. Nous déroulons ce processus en utilisant le nouveau connecteur *ORDER* définit dans l'AA de la figure 5.27.

Composition des automates La première étape dans la résolution de cette interférence est la composition des automates en passant par le format *blif* (utilisation de *mergeBlif*). L'automate résultant est donné par la figure 5.31. La stratégie de composition utilisée est *restrictive* (composition avec synchronisation). Nous utilisons les variables d'entrées $a = Sensor1.Evented\_Level$ $b = Timer3.TimeOut$ et $c = PhoneMarie.StartArro$

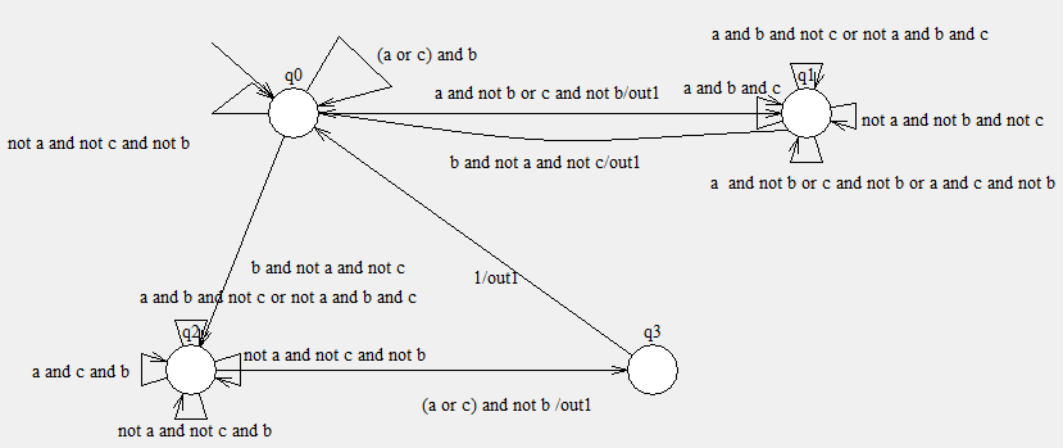


FIGURE 5.31 – L'automate résultant de la composition des automates des connecteurs *OR* et *ORDER*

De l'automate aux équations booléennes L'automate obtenu sera exploité pour déduire les équations booléennes régissant son comportement. Ces équations s'expriment à l'aide des connecteurs de base et des registres. L'automate possède 4 états donc nous avons besoin de deux registres (chapitre 3 section7).

$$X0\_Next = X0 \cdot \overline{X}1 \cdot a + X0 \cdot \overline{X}1 \cdot c + X0 \cdot \overline{X}1 \cdot \overline{b} + \overline{X}0 \cdot a \cdot \overline{b} + \overline{X}0 \cdot \overline{a} \cdot \overline{b} \cdot c$$

$$X1\_Next = \overline{X}0 \cdot X1 + \overline{X}0 \cdot \overline{a} \cdot b \cdot \overline{c}$$

$$out1 = X0 \cdot X1 + X0 \cdot \overline{a} \cdot b \cdot \overline{c} + \overline{X}0 \cdot aand \cdot b + \overline{X}0 \cdot \overline{a} \cdot \overline{b} \cdot c$$

Deux registre $D1$ et $D2$ ont été utilisés pour la gestion des états futurs ($X0\_Next$ et $X1\_Next$).

Génération de la règle de réécriture La règle de réécriture de deux connecteurs *ORDER* et *OR* est donnée par la figure 5.32. L'inconvénient de reconstruire les règles à l'aide des connecteurs de base est la complexité des règles obtenues. Une perspective (6.2.2.1) est d'essayer de trouver des règles qui s'expriment directement à l'aide des nouveaux connecteurs (*ORDER* dans ce cas).

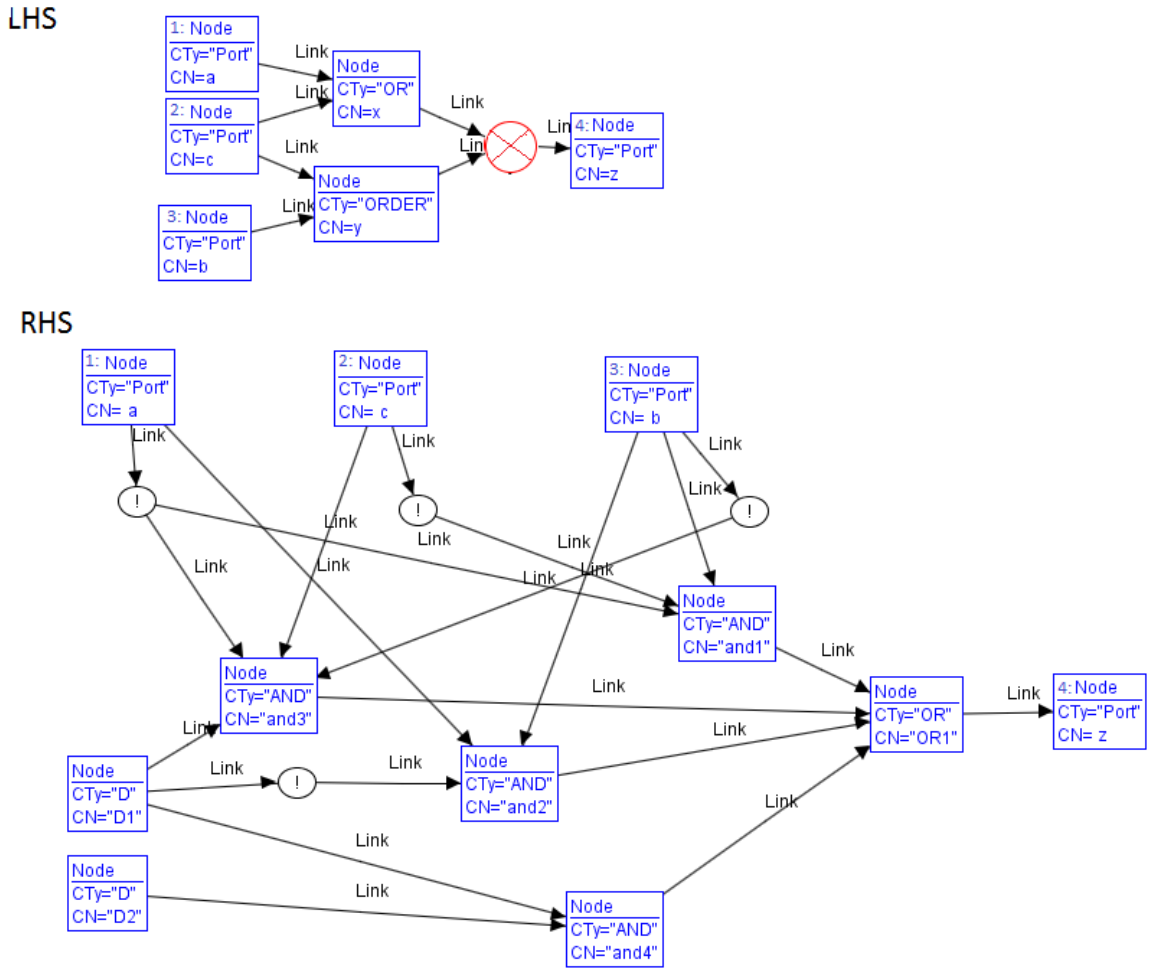


FIGURE 5.32 – La règle de transformation générée

Le résultat Le graphe résultant de toutes ces opérations sera par la suite transformé vers la plateforme WComp et prendra la forme d'un assemblage de composants illustré par la figure 5.33. Dans cet assemblage, les connecteurs et les composants sur étagère prennent la forme de boîtes avec comme image un engrenage. Les autres composants qui représentent les services pour dispositif ont une image représentant le dispositif. Par exemple, pour les comportements de gestion des lampes via la luminosité et la gestion des fenêtres selon la température, les composants de type *Threshold* seront utilisés

pour (1) récupérer la valeur depuis le capteur correspondant et (2) comparer la valeur reçue au seuil déjà défini.

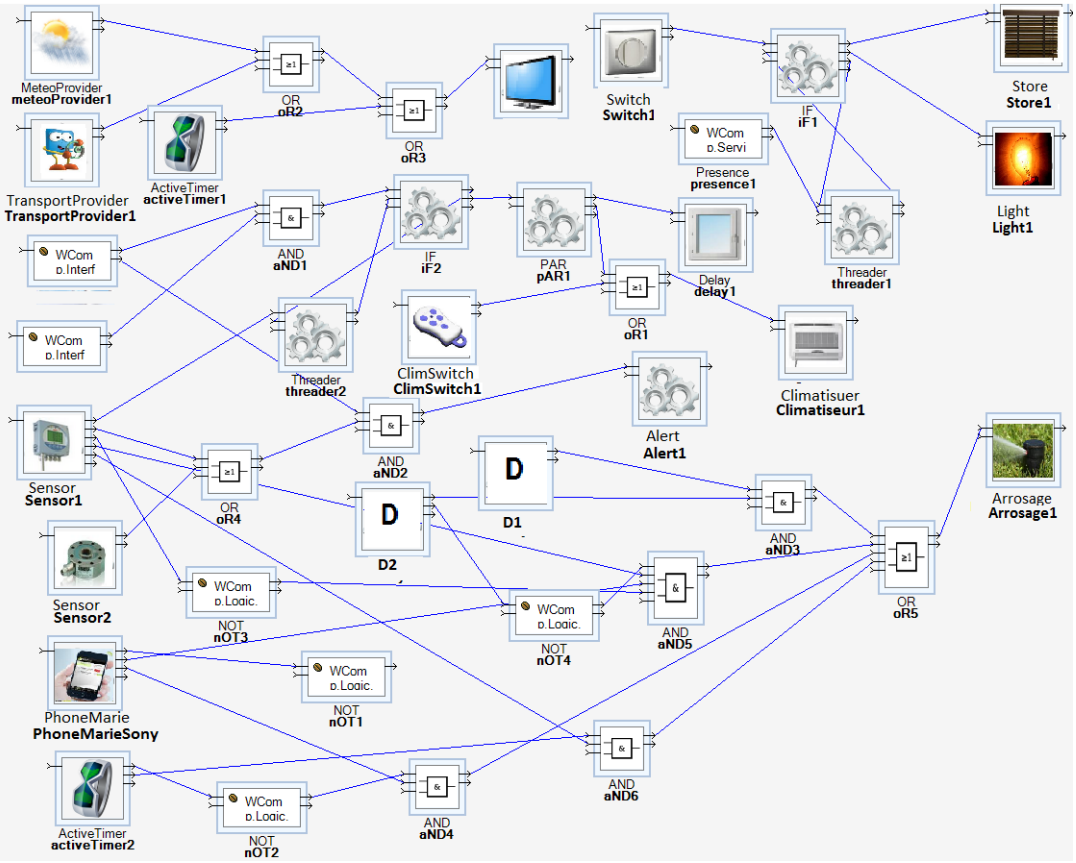


FIGURE 5.33 – Assemblage de composants WComp

5.5 Conclusion

Nous avons projeté notre modèle de graphe utilisé pour la gestion des interférences sur la plate-forme expérimentale WComp, conforme au modèle SLCA. La spécification des entités d'adaptation a été effectuée en utilisant les Aspects d'Assemblage. Cela nous a permis de spécifier des modifications réparties à travers tout un assemblage de composants. La résolution des interférences entre les AA fait appel à un ensemble de règles de réécriture de graphe qui exploitent la connaissance des connecteurs. À travers un scénario, nous avons montré le processus de composition en y incluant l'étape de la gestion des interférences. Nous avons illustré également un exemple d'ajout dynamique d'un nouveau connecteur et par la suite le processus d'extension automatique de notre mécanisme de gestion des interférences.

CHAPITRE 6

Conclusions et perspectives

Sommaire

6.1 Synthèse

Dans cette thèse, nous avons proposé un mécanisme de gestion des interférences adapté à la composition des applications en informatique ambiante. La construction des applications en IAm engage de nombreux dispositifs variés intégrés aux objets de la vie quotidienne. L'imprévisibilité de disponibilité de ces dispositifs rend le besoin d'adaptation explicite pour ce type de système. La spécificité de cette adaptation est qu'elle devra répondre à l'ensemble des contraintes imposées par le cadre de l'informatique ambiante (section 1.1.2). En particulier, il est nécessaire qu'elle respecte la propriété de boîte noire des différentes entités logicielles impliquées (dispositifs de l'espace ambiants auxquels nous n'avons pas accès hormis à travers leurs interfaces). Il s'agit alors d'une *adaptation compositionnelle* qui vise à intégrer de nouvelles entités qui n'avaient pas été prévues à la conception, de supprimer ou d'échanger les entités qui ne sont plus accessibles dans un contexte donné.

La spécification des entités d'adaptation fait appel à plusieurs designers experts de divers domaines. Chacun de ces expert détient la connaissance des entités d'adaptation qui le concerne sans forcement connaître les autres parties du système. Nous avons pu montrer que l'AOP est un mécanisme qui peut être utilisé pour la spécification des entités d'adaptation puisqu'il permet une séparation explicite des préoccupations ainsi qu'une spécification des fonctionnalités diverses et indépendantes. Cette spécification parallèle est alors à l'origine des problèmes d'interférences (2.1) qui peuvent apparaître lorsque ces entités d'adaptation sont intégrées au sein d'un même système.

Une interférence est une interaction négative entre les aspects c'est à dire qu'il existe au moins l'un de ces aspects qui ne fonctionne pas comme c'est prévu à cause de cette interaction. Il existe plusieurs types d'interférences. Nous avons montré à travers une étude de l'état de l'art que nous pouvons traiter les problèmes d'interférences d'ordre *syntaxique* au niveau d'un *point de jonction partagé*. Nous adressons particulièrement le problème de **concurrence** entre les spécifications d'adaptation. Cela signifie que quelque soit l'ordre dans lequel ils sont appliqués, le problème d'interférence persiste toujours. Sur la base de notre étude de la littérature, nous adoptons une stratégie de résolution **interne** qui fusionne les comportements fournis par les aspects dont l'objectif est la résolution des interférences. Cette fusion se base traditionnellement sur des opérateurs fournis par les langages d'aspect.

Dans notre approche ces opérateurs prennent la forme de connecteurs qui assemblent les entités logicielles disponibles. De ce fait, nous avons étudié dans la deuxième partie de notre état de l'art des approches pour la formalisation de ces connecteurs et ainsi que leur composition.

À partir de tous ces constats, nous proposons un mécanisme de gestion des interférences pour traiter les problèmes apparus lors de la composition des spécifications de compositions. Le mécanisme proposé est *dynamique* (les adaptations viennent s'appliquer sur une application qui s'exécute), *externe* (la résolution est libérée de toute contrainte liée à l'application) et *ouvert* (l'ensemble des adaptations pourra être étendu à l'exécution. Notre mécanisme est paramétré par les adaptations à composer). L'approche que nous avons définie est basée sur un formalisme de graphe, dans lequel les nœuds sont les ports des entités logicielles ou des connecteurs et les arcs représentent leurs interactions. L'utilisation du graphe comme une représentation abstraite d'une application nous a permis (1) de définir notre contribution indépendamment des modèles et des plates-formes d'implémentation et (2) d'effectuer des vérifications du résultat de la composition avant de projeter les modifications vers l'application qui s'exécute. Nous avons utiliser le formalisme de graphe pour la détection et pour la résolution des interférences. La détection des interférences vise à marquer les endroits dans le graphe où un *pattern* d'interférence a été trouvé (*matching*). La résolution de ces interférences va réécrire ce graphe en un autre où les problèmes d'interférences ont été résolus. La réécriture de graphe utilise un ensemble de règles prédéfinies. Ces règles sont fournies manuellement par un Super Designer qui se charge de la conception du mécanisme de gestion des interférences. La logique de résolution qui a été attribuée aux règles de réécriture exploite la connaissance de la sémantique des connecteurs. Une règle de réécriture est donnée pour deux connecteurs. Nous avons autant de règles que de combinaisons possible de connecteurs. Chaque règle propose de réécrire les connecteurs deux à deux de manière à faire résoudre le problème en respectant les comportements de chacun de deux connecteurs. Nous avons prouvé la propriété de symétrie de ces règles qui garantissait que le résultat sera le même quelque soit l'ordre de composition. Grâce à cette propriété, les adaptations sont indépendantes les unes des autres. De nouvelles adaptations peuvent être ajoutées dynamiquement sans se préoccuper de celles qui existent déjà. De cette manière, divers acteurs ont la possibilité de déployer au runtime leurs adaptations sans se préoccuper des problèmes d'interférences qui peuvent apparaître. Ces derniers seront traités automatiquement.

La logique de résolution que nous avons définie est fortement liée à l'ensemble des connecteurs utilisés dans la spécification des compositions. L'apparition d'une interférence utilisant un connecteur non connu à l'avance rend notre mécanisme incapable de fournir une solution. Avoir un mécanisme de gestion d'interférence extensible est particulièrement important dans le cadre de l'informatique ambiante dans la mesure où la commercialisation des nouveaux types de dispositif donne probablement naissance à la définition des nouvelles logique de coordination (donc de nouveaux connecteurs). Nous avons proposé une méthodologie permettant à notre mécanisme de gestion des interférences d'être **extensible dynamiquement et automatiquement**. Au lieu de se limiter à un ensemble fixe de connecteurs pour être capable de trouver une solution, nous proposons d'utiliser un ensemble ouvert de connecteurs. Une propriété principale est nécessaire à cette modification : *fournir une modélisation comportementale des nouveaux connecteurs*. Nous avons fait l'hypothèse que tout nouveau connecteur fournit une description de son comportement à l'aide du modèle de machine de Mealy. En effet, l'extension du mécanisme de résolution revient à incrémenter sa base de règles en rajoutant des règles permettant la réécriture de tout nouveau connecteur. Pour être extensible, il faut trouver un moyen de générer ces règles. Pour ce faire, nous avons proposé de composer l'automate du nouveau connecteur

avec celui où l'interférence a été détectée. L'automate obtenu sera transformé en équation booléenne. Cette équation est définie à l'aide des connecteurs de base (*AND*, *OR* et *NOT*) et sera exploitée pour la génération de la règle de réécriture. Cette dernière réécrit les connecteurs en interférences en utilisant des connecteurs de base (et des registres si besoin). Sous cette hypothèse, nous avons démontré qu'il est possible d'incrémenter dynamiquement notre mécanisme de gestion des interférences en ajoutant de nouveaux connecteurs.

6.2 Perspectives

L'ensemble de ces travaux de thèse ouvre de nombreuses perspectives pour des travaux futurs. Nous terminerons ce chapitre par une présentation de quelques perspectives par rapport à l'implémentation (à court terme) et aux activités de recherche (à long terme).

6.2.1 Perspectives à court terme : Connecteur Complexe

Proposition de modèle complexe

Nous avons vu que lors de la résolution d'interférences d'accès concurrent à un port d'entrée d'une entité, qu'un sous ensemble de connecteurs ont besoin d'être synchronisés. Pour ce faire, il pourrait être intéressant de tous les encapsuler dans une même entité (un connecteur complexe). Nous proposons un modèle de connecteur complexe qui est défini par le quadruplet suivant : $C_x =< P_e, p_r, \mathcal{A}_{dec}, \mathcal{L}_{fus} >$ avec :

- P_e un l'ensemble d'événements d'entrée
- Un événement de sortie p_r.
- Un automate $\mathcal{A}_{dec}$ décrivant la logique qui permet le déclenchement du l'événement de sortie p_r
- Une fonction $\mathcal{L}_{fus}$ décrivant comment les données reçues en entrée seront fusionnées.

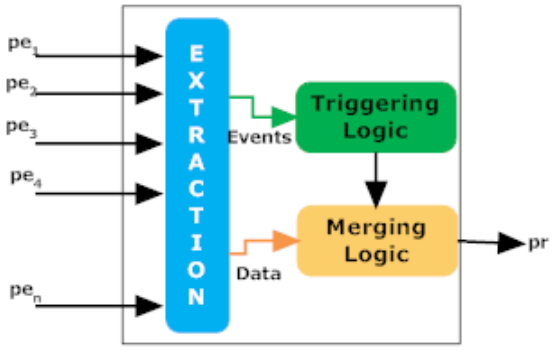


FIGURE 6.1 – Modèle du connecteur complexe

La Figure 6.1 montre la structure interne d'un connecteur complexe synchrone. La boîte *Extraction* est utilisée pour séparer la partie données des événements reçus en entrée. Elle transmet vers la boîte *Triggering Logic* un ensemble d'événements purs. La partie Triggering Logic ($\mathcal{A}_{dec}$) détermine si un événement de sortie p_r va être émis. Dans ce cas, la boîte *Merging Logic* va être activée par la suite pour déterminer quelle donnée envoyer avec l'événement de sortie. Dans cette thèse, nous n'abordons pas le problème de la fusion de données. Dans notre approche, nous nous intéressons uniquement à la partie logique de déclenchement. La partie de fusion de données sera

une perspective de recherche. Pour une première implémentation nous considérons que tous les connecteurs possèdent la même logique de fusion des données.

Dans la figure 6.2, nous présentons un exemple d'accès concurrent au port *Disp.m1*. La fusion des comportements de ces deux connecteurs donne naissance à un connecteur complexe qui possède une logique de déclenchement $A3$. La logique de déclenchement d'un connecteur complexe est un assemblage de connecteurs primitifs (par exemple les connecteurs *AND*, *OR*). Cet assemblage reflète une composition d'automates de chacun de ces connecteurs primitifs. L'ensemble des ports d'entrée d'un connecteur complexe possède une dynamique différente et les événements n'arrivent pas tous en même temps voire même dans un ordre inconnu à l'avance. Il sera nécessaire de gérer la synchronisation qui est la fonction principale d'un connecteur complexe.

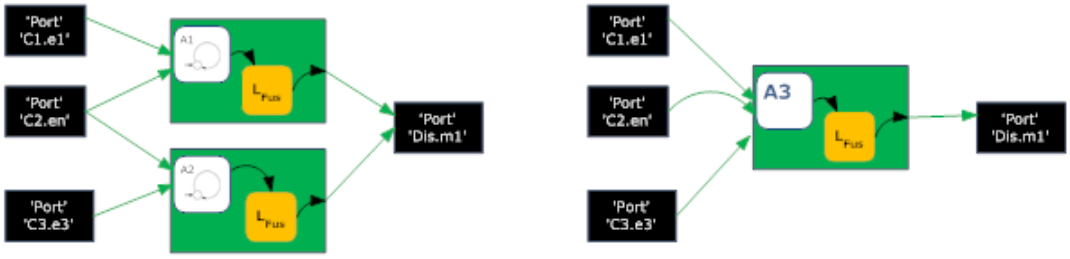


FIGURE 6.2 – Fusion de deux connecteurs pour l'obtention d'un nouveau connecteur complexe

Machine d'exécution

Pour passer d'un système synchrone à son implémentation effective et confronter la nature asynchrone du monde réel, il faut définir une machine d'exécution pour les connecteurs complexes. Le système synchrone doit être enfoui dans son environnement en assurant une approximation de l'hypothèse synchrone. Dans un cas réel, l'hypothèse de temps de réaction nul est non réaliste. Le principe d'**atomicité** des réactions peut rapprocher l'hypothèse d'instantanéité des réactions. L'atomicité engendre deux faits : (1) figer la perception de l'environnement pendant la réaction et (2) la réaction est considérée comme un bloc qui doit être entamé en totalité ou bien complètement annulé. Il sera indispensable de vérifier que le système est assez rapide vis-à-vis de la dynamique de son environnement. Il est indispensable aussi de faire le lien entre le temps logique et le temps physique ainsi que le lien entre les événements réels de l'environnement et ceux utilisés dans la définition du processus synchrone. L'entité qui se charge d'assurer ces approximations est **la machine d'exécution** et permet de soigner l'interfaçage entre le processus synchrone et l'environnement asynchrone.

Dans la littérature diverses machines d'exécution ont été proposées. Dans sa thèse, *Boufeid* [Bou98] a proposé une machine d'exécution selon le modèle de boîtes réactives. Le modèle proposé est assez générique et s'applique à plusieurs langages synchrones. Ce modèle permet de décrire les différentes stratégies pour gérer le parallélisme entre l'environnement et le système synchrone. L'interaction entrées/sorties dépend de l'activation du comportement réactif du processus synchrone. Ces activations sont déterminées par l'environnement. Il est important de savoir quand une réaction est déclenchée et quand celle-ci est terminée. Pour cette raison deux signaux de contrôle BoI et EoI ont été définis. Cela se base sur l'hypothèse de l'exclusion entre l'évolution de l'environnement et l'évolution de la machine. Cependant, dans la réalité, il faut considérer le parallélisme qui existe entre

la machine et l'environnement car des événements peuvent apparaître durant la réaction et devront être pris en considération. Le principe d'atomicité rend difficile l'annulation de la réaction pour considérer l'apparition de nouveaux événements. La solution proposée par *Boufeid* est de mémoriser les événements qui se produisent pendant une réaction et les utiliser pendant la réaction suivante. Ceci est rendu possible grâce à l'utilisation des contrôleurs d'entrée et des contrôleurs de sortie qui se chargent de gérer les occurrences des événements et de faire le lien entre les événements du monde réel et les événements logiques utilisés par le processus synchrone. A cause de la variété des sources d'événements du monde réel, à chaque événement d'entrée/sortie est associé un contrôleur individuel pour faire le couplage nécessaire entre le mode synchrone et asynchrone (conversion de type, etc.). Les contrôleurs d'entrée sont tous regroupés dans un seul module pour former le Input Module (figure 6.3.a) ; c'est-à-dire qu'à partir des événements d'entrée, il forme l'image instantanée de l'environnement sous forme d'un vecteur I.

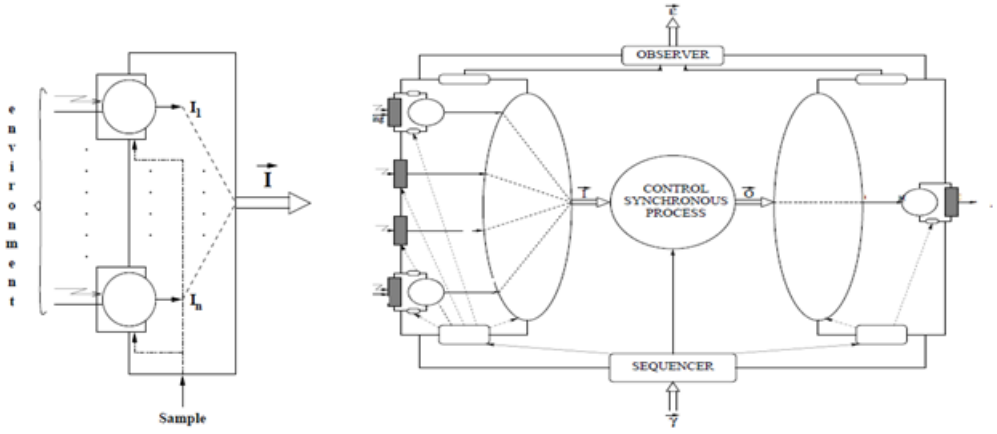


FIGURE 6.3 – a) Input Module (b) Modèle complet de la machine d'exécution

Dans le même esprit, l'ensemble des signaux de sortie sont regroupés dans un *outputModule* qui transforme le vecteur de sortie O en des actions sur l'environnement. La figure 6.3.b illustre la machine d'exécution proposée et qui considère trois phases :

- Suite à la réception du signal *BoI*, il y aura l'activation du module d'input pour la construction du vecteur d'entrée I
- Activation du processus synchrone : Obtention du vecteur O
- Activation de l'output Module : action sur l'environnement et émission du signal *EoI*

Pour traiter certains cas de dysfonctionnement, *Boufeid* a ajouté la boîte *OBSERVER* qui détecte des cas d'anomalie et remonte ces informations (réaction très lente du processus synchrone, acquisition de plusieurs occurrence d'un même événement pendant une réaction, etc.). A ce niveau, des solutions peuvent être associées à certaines anomalies. Par exemple, on pourra décider d'écraser l'ancienne occurrence d'un évènement et ne laisser que l'occurrence la plus récente. La boîte *SEQUENCER* définit la sémantique d'exécution de la machine d'exécution. Cette boîte va assurer l'enchaînement en trois phases de la machine. Elle va assurer aussi la manière dont une réaction peut être démarrée, interrompue (l'interruption est autorisé seulement pendant la phase 2), suspendue (ensuite reprise) ou bien réinitialisée.

6.2.2 Perspectives à long terme

Les perspectives par rapport aux activités de recherche sont multiples.

6.2.2.1 Règles de réécriture exprimées à l'aide du nouveau connecteur

Nous avons vu que le mécanisme de gestion d'interférences proposé repose sur un ensemble de connecteurs de base dont la composition deux à deux a été prouvée comme étant symétrique. Malgré la possibilité d'extension dynamique de cet ensemble de connecteurs, la solution proposée par notre mécanisme n'utilise que les connecteurs de base et non plus les nouveaux connecteurs. Cela signifie que pour un nouveau connecteur New_C à composer avec un autre connecteur $Conn_i$: $New_C \otimes Conn_i = F(AND, OR, NOT)$. Il serait intéressant, dans des travaux futurs, d'améliorer ce mécanisme de composition. Un premier axe de travail consisterait à exprimer cette composition à l'aide de nouveaux connecteurs, c'est à dire : $New_C \otimes Conn_i = F(New_C, conn_i)$. Pour faciliter cette tâche, il serait profitable de disposer d'un mécanisme permettant d'extraire à partir d'un automate le comportement relatif de chacun de nos connecteurs pour pouvoir par la suite trouver leur règle de réécriture.

6.2.2.2 Réduire le temps de réponse

Le temps nécessaire pour adapter le système aux variations de l'infrastructure doit être pris en compte afin d'améliorer l'expérience utilisateur et de faire des applications adaptées et cohérentes avec leur infrastructure logicielle réelle. Dans ce contexte, le défi du contrôle de la durée de ce processus d'adaptation a été abordé dans [TLR+12]. Cependant, l'extension du mécanisme de gestion des interférence que nous proposons est plus complexe (utilisation des graphes) et a fait augmenter le temps de réponse de notre processus d'adaptation. Pour tenter de réduire ce temps de réponse et retrouver les performances nécessaires dans le cadre de l'informatique ambiante nous proposons deux pistes :

Pré fusion statique L'idée est de ramener une partie du problème au design time. Le problème d'interférence se produit lors de la composition de plusieurs instances d'adaptations. La pré fusion propose de calculer la composition de toutes les combinaisons possible de spécification de composition. Cela nécessite de travailler au niveau type d'entités logicielles et non au niveau des instances. Pour ce faire, il faudra avoir un mécanisme qui analyse les points de coupe des aspects d'assemblage pour détecter les interférences potentielles. Soit $\mathcal{A}$ l'ensemble des Aspects d'assemblage, $\mathcal{A} = \{AA_0, AA_1, ..., AA_n\}$. Un aspect d'assemblage $AA_i = (pointcut_i, advice_i)$ est en interférence avec un autre aspect $AA_j = (pointcut_j, advice_j)$ s'il existe une relation entre $pointcut_i$ et $pointcut_j$ c'est-à-dire, il existe $Rule_{ik} \in pointcut_i$ et $Rule_{jz} \in pointcut_j$ tel que $\mathcal{R}(Rule_{ik}, Rule_{jz})$ est vrai. Deux types de relations entre les règles formant les points de coupe sont possibles : *Inclusion* et *Identique*. Une règle d'un point de coupe est identique à une autre règle si elle permet de sélectionner exactement le même ensemble de points de jonction. L'inclusion signifie que cette règle identifie une partie des points de jonction d'une autre. La technique utilisée dans ce cas est forcement liée au langage utilisé pour exprimer les points de coupe.

Incrémentalité et décrementalité Dans notre solution actuelle, suite à l'apparition ou la disparition d'un dispositif, nous recalculons à partir de l'application de base toutes les adaptations possibles même si elles ne sont pas concernées par la variation. Nous proposons de rendre le mécanisme de composition incrémentale (respectivement décrementale). Cela signifie que suite à l'apparition d'un nouveau dispositif à un instant t, nous utilisons le graphe du système obtenu à l'instant

$t-1$ auquel nous rajoutons les nouvelles adaptations à composer. Dans le cas d'une disparition d'un dispositif, il faudra avoir un mécanisme permettant de calculer le sous-graphe à retirer du graphe de l'application pour retrouver le comportement sans ces dispositifs (retirer les adaptations qui ont été activées par ce dispositif).

6.3 Liste des publications

Revues internationales

Jean Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Nicolas Ferry, Vincent Hourdin, **Sana Fathallah**, Christophe Vergoni, Michel Riveill.*"Aspects of Assembly : from Theory to Performance"*, in Transactions on Aspect Oriented Software Development (TAOSD), vol. 7271, Springer, 2012.

Conférences internationales

Sana Fathallah, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "*A Symmetric Compositional Approach for Adaptive Ubiquitous Systems*" in Proceedings of the 15th IEEE International Conference on Computational Science and Engineering (CSE), IEEE Computer Society, Paphos, Chypre, 5-7 December 2012

Sana Fathallah, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "*The Dynamic Composition of Independent Adaptations including Interferences Management*" in Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA), Lisbon, Portugal, 18-23 November 2012. Best Paper

Sana Fathallah, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "*Adaptations Interferences Detection and Resolution with Graph-Transformation Approach*" in Proceedings of the Proceedings of the 6th International Conference : Sciences of Electronic, Technologies of Information and Telecommunications (SETIT), Sousse, Tunisia, 21-24 March 2012 (Acceptation Rate : 36%)

Sana Fathallah, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "*A Dynamic mechanism for solving Interference Adaptation in Ubiquitous Computing Environment*" in Proceedings of the 1st International Workshop on Dynamicity (DYNAM), Toulouse, France, 12 December 2011

Sana Fathallah, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "*MergeIA : A Service for Dynamic Merging of Interfering Adaptations in Ubiquitous System*" in Proceedings of the Proceedings of the Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM), Lisbon, Portugal, 20–25 November 2011

Conférence Nationale

Sana Fathallah, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "*Résolution des interférences entre les adaptations par transformations de graphes*" in Proceedings of the 29ème

INFORSID, Lille, France, 24-26 May 2011

Rapports de recherche et livrables

Gaëtan Rey, Jean-Yves Tigli, Stéphane Lavirotte, Nicolas Ferry, **Sana Fathallah**, Joëlle Coutaz, Emeric Fontaine, Fabrice Jouanot, Marie-Christine Rousset, Philippe Renevier, Anne-Marie Pinna-Déry, Vincent Hourdin. "*Modélisation du contexte et Adaptation*" Research Report ANR Continuum, number D2.1-2.2, 1-57 pages, August 2010

Sana Fathallah, Stéphane Lavirotte, Jean-Yves Tigli, Kamel Hamrouni. "*Tissages Multiples d'Aspects d'Assemblage : Application à l'adaptation logicielle pour l'Informatique Ambiante*" Research Report I3S (Université de Nice - Sophia Antipolis / CNRS), number I3S/RR-2009-16-FR, 44 pages, Sophia Antipolis, France, November 2009

Bibliographie

[A+05] Farhad Arbab et al. Composition by interaction. *Leiden University*, 2005.

[All97] Robert J Allen. A formal approach to software architecture. Technical report, DTIC Document, 1997.

[AM02] Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. In *Coordination Models and Languages*, pages 22–39. Springer Berlin Heidelberg, 2002.

[Arb98] Farhad Arbab. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, 1122, 1998.

[Arb03] Farhad Arbab. Abstract behavior types : A foundation model for components and their composition. In *Formal Methods for Components and Objects*, pages 33–70. Springer, 2003.

[Arn92] André Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson, 1992.

[ARS09] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 39–50. ACM, 2009.

[AZI+08] Dionysis Athanasopoulos, Apostolos V Zarras, Valerie Issarny, Evaggelia Pitoura, and Panos Vassiliadis. Cowsami : Interface-aware context gathering in ambient intelligence environments. *Pervasive and Mobile Computing*, 4(3) :360–389, 2008.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software : Practice and Experience*, 36(11-12) :1257–1284, 2006.

[Ber01] Laurent Berger. *Mise en oeuvre des interactioons en environnements distribués, compilés et fortement typés : le modèle MICADO*. PhD thesis, Université de Nice Sophia Antipolis, 2001.

[Bou98] Hedi Boufaied. *Machines d'execution pour langages synchrones*. PhD thesis, Université de Nice Sophia Antipolis, 1998.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of Computer Programming*, 61(2) :75–113, 2006.

[CBG+08] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems (TOCS)*, 26(1) :1, 2008.

[CFW09] Daniel Cheuing Foo Wo. *Adaptation dynamique par tissage d'aspects d'assemblage*. PhD thesis, Université de Nice Sophia Antipolis, 2009.

[CHA+10] Selim Ciraci, Wilke Havinga, Mehmet Aksit, Christoph Bockisch, and Pim van den Broek. A graph-based aspect interference detection approach for uml-based aspect-oriented models. In *Transactions on aspect-oriented software development VII*, pages 321–374. Springer, 2010.

[Cho56] Noam Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3) :113–124, 1956.

[CLF05] Tarak Chaari, Frédérique Laforest, and André Flory. Adaptation des applications au contexte en utilisant les services web. In *Proceedings of the 2nd French-speaking conference on Mobility and ubiquity computing*, pages 111–118. ACM, 2005.

[CM04] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with ao4bpel. In *Web Services*, pages 168–182. Springer, 2004.

[CWI13] CWI. General description of a connector@ONLINE, Juin 2013.

[DBA07] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Static and dynamic detection of behavioral conflicts between aspects. In *Runtime Verification*, pages 38–50. Springer, 2007.

[Dey00] Anind K Dey. *Providing architectural support for building context-aware applications*. PhD thesis, Georgia Institute of Technology, 2000.

[DFL+06] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *Transactions on Aspect-Oriented Software Development I*, pages 174–213. Springer, 2006.

[DFQJ08] Bruno De Fraine, Pablo Daniel Quiroga, and Viviane Jonckers. Management of aspect interactions using statically verified control flow relations. In *Proceedings of the Third International Workshop on Aspects, Dependencies and Interactions (held at ECOOP)*, pages 5–14. Citeseer, 2008.

[DK12] Cynthia Disenfeld and Shmuel Katz. A closer look at aspect interference and cooperation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 107–118. ACM, 2012.

[DL+06] Pierre-Charles David, Thomas Ledoux, et al. Une approche par aspects pour le développement de composants fractal adaptatifs. *L'OBJET*, 12(2-3) :113–132, 2006.

[DMB09] Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. The art of the meta-aspect protocol. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 51–62. ACM, 2009.

[DS02] Rémi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (eaop). *Techn. Ber., Ecole des Mines de Nantes. TR*, 2(11), 2002.

[EKL91] Hartmut Ehrig, Martin Korff, and Michael Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In *Graph grammars and their application to computer science*, pages 24–37. Springer, 1991.

[Eng97] Robert Englander. *Developing Java Beans*. Oreilly, 1997.

[ENRR88] Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld. *Graph-grammars and Their Application to Computer Science : 3rd International Workshop, Warrenton, Virginia, USA, December 2-6, 1986*, volume 291. Springer, 1988.

[Fat09] Sana Fathallah. Tissages Multiples d'Aspects d'Assemblage : Application à l'adaptation logicielle pour l'Informatique Ambiante. Master's thesis, Université de Tunis El Manar, Tunis, Tunisie, 2009.

[FBALT+12a] Sana Fathallah Ben Abdenneji, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, and Michel Riveill. The dynamic composition of independent adaptations including interferences management. In *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, pages 678–684, 2012.

[FBALT+12b] Sana Fathallah Ben Abdenneji, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, and Michel Riveill. A symmetric compositional approach for adaptive ubiquitous systems. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pages 223–228. IEEE, 2012.

[FBBLT+11] Sana Fathallah Ben Bbdenneji, Stephane Lavirotte, Jean-Yves Tigli, Gaetan Rey, and Michel Riveill. Mergeia : A service for dynamic merging of interfering adaptations in ubiquitous system. In *UBICOMM 2011, The Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 34–38, 2011.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM, 2004.

[Fer11] Nivolas Ferry. *Adaptations dynamiques au contexte en informatique ambiante : propriétés logiques et temporelles*. PhD thesis, Université de Nice Sophia Antipolis, 2011.

[FLT+12] Sana Fathallah, Stephane Lavirotte, Jean-Yves Tigli, Gaetan Rey, and Michel Riveill. Adaptations interferences detection and resolution with graph-transformation approach. In *Sciences of Electronics, Technologies of Information and Telecommunications (SETIT), 2012 6th International Conference on*, pages 36–43. IEEE, 2012.

[Gaf12] Daniel Gaffe. Suite logicielle autour des machines d'Ã©tats finis@ONLINE, Decembre 2012.

[Gel85] David Hillel Gelernter. *Parallel programming in Linda*. Yale University, Department of Computer Science, 1985.

[GLS+07] P Greenwood, B Lagaisse, F Sanen, G Coulson, A Rashid, E Truyen, and W Joosen. Interactions in ao middleware. In *Proc Workshop on ADI, ECOOP*, 2007.

[GLTJ08] Paul Grace, Bert Lagaisse, Eddy Truyen, and Wouter Joosen. A reflective framework for fine-grained adaptation of aspect-oriented compositions. In *Software Composition*, pages 215–230. Springer, 2008.

[GMW10] David Garlan, Robert Monroe, and David Wile. Acme : an architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173. IBM Corp., 2010.

[GRWK09] Kurt Geihs, Roland Reichle, Michael Wagner, and Mohammad Ullah Khan. Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. In *Software engineering for self-adaptive systems*, pages 146–163. Springer, 2009.

[Gue06] Mohammed Karim Guennoun. *Architectures dynamiques dans le contexte des applications à base de composants et orientées service*. PhD thesis, Université Paul Sabatier-Toulouse III, 2006.

[Hav09] WK Havinga. On the design of software composition mechanisms and the analysis of composition conflicts. 2009.

[HDA11] Abdelhakim Hannousse, Rémi Douence, and Gilles Ardourel. Static analysis of aspect interaction and composition in component models. In *ACM SIGPLAN Notices*, volume 47, pages 43–52. ACM, 2011.

[HJ06] MA Zhi-Yi Chen Hong-Jie. A service-oriented architecture reference model. *Chinese Journal of Computers*, 7 :002, 2006.

[HL95] Walter L Hürsch and Cristina Videira Lopes. Separation of concerns. 1995.

[HLH08] Chengwan He, Zheng Li, and Keqing He. Towards trusted aspect composition. In *Computer and Information Technology Workshops, 2008. CIT Workshops 2008. IEEE 8th International Conference on*, pages 643–648. IEEE, 2008.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[Hou10] Vincent Hourdin. *Contexte et sécurité dans les intergiciels d'informatique ambiante*. PhD thesis, Université de Nice Sophia Antipolis, 2010.

[JGP00] E. M. Clarke Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[Kar53] Maurice Karnaugh. The map method for synthesis of combinational logic circuits. *American Institute of Electrical Engineers, Part I : Communication and Electronics, Transactions of the*, 72(5) :593–599, 1953.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001 Object Oriented Programming*, pages 327–354. Springer, 2001.

[KK08] Emilia Katz and Shmuel Katz. Incremental analysis of interference among aspects. In *Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 29–38. ACM, 2008.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[LA07] Alexander Lazovik and Farhad Arbab. Using reo for service coordination. In *Service-Oriented Computing–ICSOC 2007*, pages 398–403. Springer, 2007.

[LEW05] Kung-Kiu Lau, Perla Velasco Elizondo, and Zheng Wang. Exogenous connectors for software components. In *Component-Based Software Engineering*, pages 90–106. Springer, 2005.

[LLUE07] Kung-Kiu Lau, Ling Ling, Vladyslav Ukis, and Perla Velasco Elizondo. Composite connectors for composing software components. In *Software Composition*, pages 266–280. Springer, 2007.

[LUVW06] Kung-Kiu Lau, Vladyslav Ukis, Perla Velasco, and Zheng Wang. A component model for separation of control flow from computation in component-based systems. *Electronic Notes in Theoretical Computer Science*, 163(1) :57–69, 2006.

[LWF03] Antónia Lopes, Michel Wermelinger, and José Luiz Fiadeiro. Higher-order architectural connectors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1) :64–104, 2003.

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. In *Advanced lectures on networking*, pages 20–58. Springer, 2002.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 3–14. ACM, 1996.

[MMP00] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187. ACM, 2000.

[MMT06] Katharina Mehner, Mattia Monga, and Gabriele Taentzer. Interaction analysis in aspect-oriented models. In *Requirements Engineering, 14th IEEE International Conference*, pages 69–78. IEEE, 2006.

[Mos10] Sébastien Mosser. *Behavioral composition in service oriented architecture*. PhD thesis, Université de Nice Sophia Antipolis, 2010.

[MSA06] Mohammad Reza Mousavi, Marjan Sirjani, and Farhad Arbab. Formal semantics and analysis of component connectors in reo. *Electronic Notes in Theoretical Computer Science*, 154(1) :83–99, 2006.

[MSKC04] Philip K McKinley, Seyed Masoud Sadjadi, Eric P Kasten, and Betty HC Cheng. A taxonomy of compositional adaptation. *Rapport Technique numéroMSU-CSE-04-17, juillet*, 2004.

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152 :125–142, 2006.

[MW09] Antoine Marot and Roel Wuyts. Detecting unanticipated aspect interferences at runtime with compositional intentions. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, page 3. ACM, 2009.

[OGT+99] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE*, 14(3) :54–62, 1999.

[PGA01] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic homogenous aop with prose. *Switerland, Department of Computer Science, ETH Zürich*, 2001.

[PTDL07] Mike P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing : State of the art and research challenges. *Computer*, 40(11) :38–45, 2007.

[RE99] Grzegorz Rozenberg and Hartmut Ehrig. *Handbook of graph grammars and computing by graph transformation*, volume 1. World Scientific London, 1999.

[Rei85] Wolfgang Reisig. *Petri nets : an introduction*. Springer-Verlag New York, Inc., 1985.

[RTL+10] Gaëtan Rey, Jean-Yves Tigli, Stéphane Lavirotte, Nicolas Ferry, Sana Fathallah, Joëlle Coutaz, Emeric Fontaine, Fabrice Jouanot, Marie-Christine Rousset, Philippe Renevier, Anne-Marie Pinna-Déry, and Vincent Hourdin. Modélisation du contexte et Adaptation. Technical Report D2.1-2.2, ANR Continuum, August 2010.

[SAM06] David Stauch, Karine Altisen, and Florence Maraninchi. Interference of larissa aspects. In *FOAL 2006 Proceedings*, page 57, 2006.

[Sat01] Mahadev Satyanarayanan. Pervasive computing : Vision and challenges. *Personal Communications, IEEE*, 8(4) :10–17, 2001.

[SDZ96] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and implementations for architectural connections. In *Configurable Distributed Systems, 1996. Proceedings., Third International Conference on*, pages 2–10. IEEE, 1996.

[SFBK+12] Meng Sun, Arbab Farhad, Aichernig Bernhard K., Astefanoaei Lcramioara, and Rutten Jan. Connectors as designs : Modeling, refinement and test case generation. *Sci. Comput. Program.*, 77(7-8) :799–822, 2012.

[SG03] Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 374–384. IEEE, 2003.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software : beyond object-oriented programming*. Addison-Wesley, 2002.

[SSK+07] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel. A survey on aspect-oriented modeling approaches. *Relatorio tecnico, Vienna University of Technology*, 2007.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software : Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2) :14, 2009.

[STJ+06] Frans Sanen, Eddy Truyen, Wouter Joosen, Andrew Jackson, Andronikos Nedos, Siobhan Clarke, Neil Loughran, and Awais Rashid. Classifying and documenting aspect interactions. In *proceedings of the fifth AOSD workshop on aspects, components, and patterns for infrastructure software*, pages 23–26. Bonn, Germany : Published as University of Virginia Computer Science Technical Report CS–2006–01, 2006.

[Tae00] Gabriele Taentzer. Agg : A tool environment for algebraic graph transformation. In *Applications of Graph Transformations with Industrial Relevance*, pages 481–488. Springer, 2000.

[TBB04] Francis Tessier, Mourad Badri, and Linda Badri. A model-based detection of conflicts between crosscutting concerns : Towards a formal approach. In *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, 2004.

[TLR+12] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Nicolas Ferry, Vincent Hourdin, Sana Fathallah Ben Abdenneji, Christophe Vergoni, and Michel Riveill. Aspect of assembly : from theory to performance. In *Transactions on Aspect-Oriented Software Development IX*, pages 53–91. Springer, 2012.

[VEL10] Perla Velasco Elizondo and Kung-Kiu Lau. A catalogue of component connectors to support development with reuse. *Journal of Systems and Software*, 83(7) :1165–1178, 2010.

[Wei91] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3) :94–104, 1991.

[Wei93] Mark Weiser. Ubiquitous computing. *Computer*, 26(10) :71–72, 1993.

[WJ08] Jon Whittle and Praveen Jayaraman. Mata : A tool for aspect-oriented modeling based on graph transformation. In *Models in Software Engineering*, pages 16–27. Springer, 2008.

[WTR07] Nathan Weston, Francois Taiani, and Awais Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. In *Proceedings of the Workshop on Methods, Models and Tools for Fault Tolerance (MeMoT)*, pages 95–102, 2007.

[ZCVDBG06] Jing Zhang, Thomas Cottenier, Aswin Van Den Berg, and Jeff Gray. Aspect interference and composition in the motorola aspect-oriented modeling weaver. In *MoDELS*, 2006.

Quatrième partie

Annexes

ANNEXE A

Les détails des règles de réécriture

A.1 Règles de transformations de graphe

Durant la phase d'identification des interférences, nous avons marqué par un nœud particulier de type $\otimes$ les endroits où il y a des interférences. L'approche que nous avons choisie pour la résolution de ces interférences est la réécriture des connecteurs. Ceci est rendu possible par la connaissance de la sémantique d'éléments clés dans la spécification des règles d'adaptation. Donc, si nous avons plusieurs comportements (connecteurs) à la sortie d'un nœud, le mécanisme de résolution produira un comportement final qui regroupe tous les comportements spécifiés sans avoir d'interférence dans le comportement résultant. De plus, ce mécanisme de réécriture garantit la symétrie via trois propriétés : l'idempotence, la commutativité et l'associativité (section 4.1.4). De ce fait, l'ordre d'application des adaptations, donc l'ordre dans lequel la réécriture des comportements est réalisé n'a pas d'importance (le résultat ne dépend pas de cet ordre). Nous avons défini un ensemble de règles qui dérivent de [CFW09] et [Ber01]. Dans la suite de cette section nous détaillons les règles de réécriture de connecteurs deux à deux.

La réécriture de deux appels à un port Un port d'une entité logicielle pourra avoir plusieurs interactions avec des ports d'autres entités. Il s'agit d'une interférence entre des envois d'un message. La réécriture d'un envoi de message avec lui-même est décrite par la règle de la figure A.1. Dans le graphe résultant *R* un seul lien vers le port en question sera maintenu.

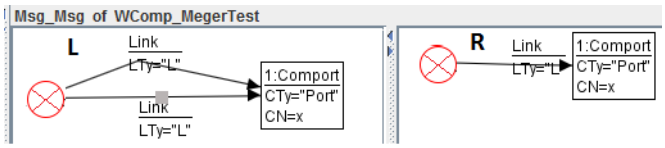


FIGURE A.1 – La Règle de fusion de deux appels d'un Port

La réécriture du PAR avec un Port Le connecteur *PAR* est utilisé pour définir que deux actions s'effectuent en parallèle. La réécriture d'un *PAR* avec un port qui appartient à l'ensemble de ses successeurs produit le graphe *R* de la figure A.2. Un seul lien (qui part du *PAR*) sera gardé vers ce port. Par exemple le résultat de fusion de : $PAR(a,B) \otimes a$ est $PAR(a,B)$ avec : $CTy(a) =$ "*Port*" et *B* représente un autre comportement (un autre port ou connecteurs).

La réécriture du PAR avec tous les autres comportements La règle de réécriture d'un connecteur concurrentiel avec un autre comportement est décrite dans la figure A.3. L'arc libellé par la chaîne de caractère "*P*" montre le sens de propagation de l'opérateur de fusion $\otimes$. Il est calculé par la fonction Pivot. Cette fonction permet de contrôler la manière dont la réécriture s'opère. Le pivot sert à guider la réécriture afin qu'elle se fasse en des points précis. Cette fonction recherche s'il existe des ports

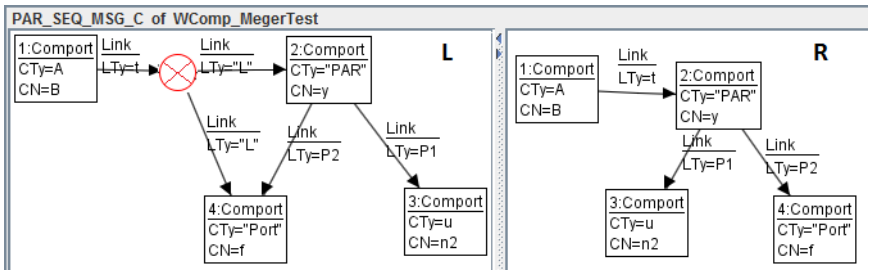


FIGURE A.2 – La Règle de fusion du PAR et un port

successeurs du *PAR* et qui ont aussi un prédécesseur le nœud $CN = x$ de la règle A.3. Si c'est le cas, une nouvelle réécriture sera appliquée entre les nœuds $CN = x$ et $CN = f$ (il a été identifié par la fonction pivot).

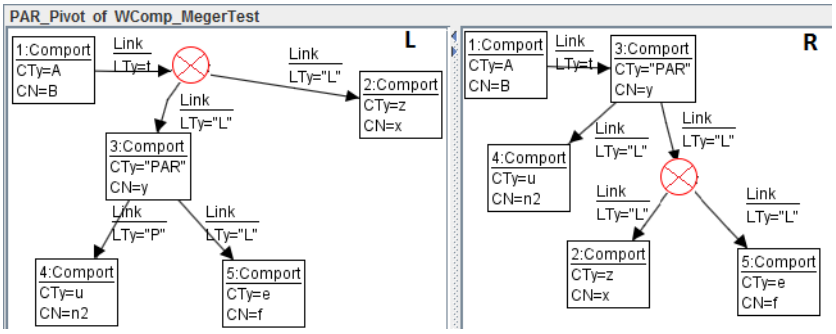


FIGURE A.3 – La Règle de fusion générique du PAR

La réécriture de deux IF Un connecteur *IF* permet de conditionner l'exécution d'une actions. Il est sous la forme $IF(m,A,B)$; avec m la condition à vérifier, A le comportement à mettre en place si m est vrai et B le comportement si m est faux. La réécriture de deux connecteurs *IF* est régie par plusieurs règles de fusion.La réécriture de $IF(m1,A,B) \otimes IF(m2,C,D)$ dépend de $m1$ et $m2$. S'il s'agit de la même condition, c'est-à-dire $m1 = m2$, alors nous gardons un seul connecteur IF et la réécriture sera propagée sur les deux branches de ce dernier (Figure A.4). Nous aurons comme résultat $IF(m1,A \otimes C,B \otimes D)$. En effet puisque nous avons la même condition à vérifier, il s'agit d'un seul *IF* d'où la fusion des deux branches vraies et des deux branches fausses de deux *IF* : c'est la propagation de l'opérateur de fusion. Par la suite la réécriture des comportements $A \otimes C$ et $B \otimes D$ fait appel à d'autres règles de réécriture. Dans l'autre cas c'est-à-dire si $m1$!=$m2$ le résultat de réécriture sera alors IF(m1,IF(m2,A⊗ C,A⊗ D),IF(m2,B⊗ C,B⊗ D)).

La règle précédente A.4 montre le cas général de la réécriture de deux IF. Une autre configuration possible est le cas où les deux connecteurs conditionnels partagent un nœud illustré dans la figure A.5. Logiquement les deux connecteurs doivent partager un nœud de la même branche (puisqu'il s'agit de la même condition à vérifier). La variable $P1$ est utilisée pour désigner soit *"T"* ou bien *"F"*. De cette manière cette règle sera appliquée si les nœuds en communs sont présents sur la branche true ou bien false de ces connecteurs.

La réécriture d'un IF et un Port Les figures A.6, A.7 et A.8 montrent que le comportement conditionnel «encapsule» le comportement séquentiel et les envoie de messages. Ces derniers seront réécrits

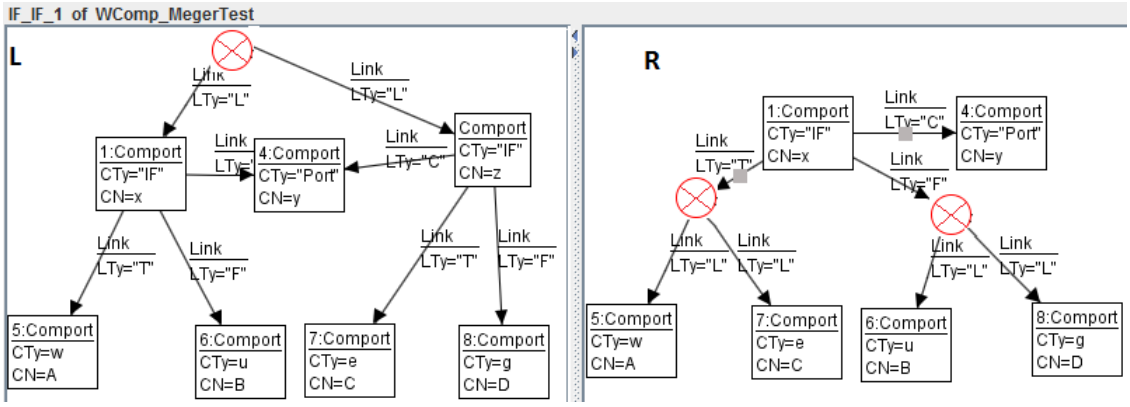


FIGURE A.4 – La Règle de fusion de deux IF : cas général

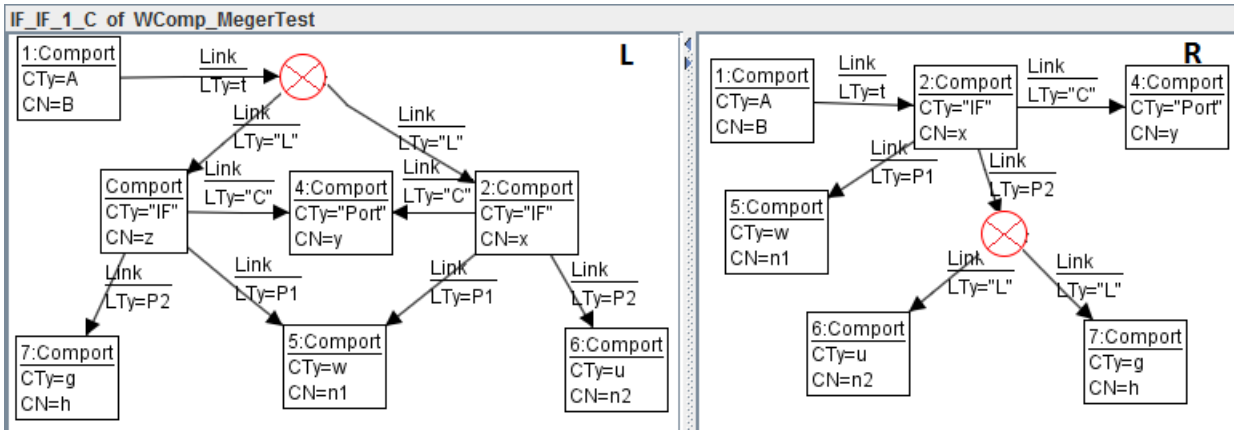


FIGURE A.5 – La Règle de fusion de deux IF : cas particulier

avec les deux branches du comportement conditionnel. La réécriture d'un *IF* avec un port présente deux cas. Si le port appartient à l'un de deux branche de IF, alors la règle de la figure A.6 sera appliquée sinon c'est la règle de la figure A.7 qui sera activée.

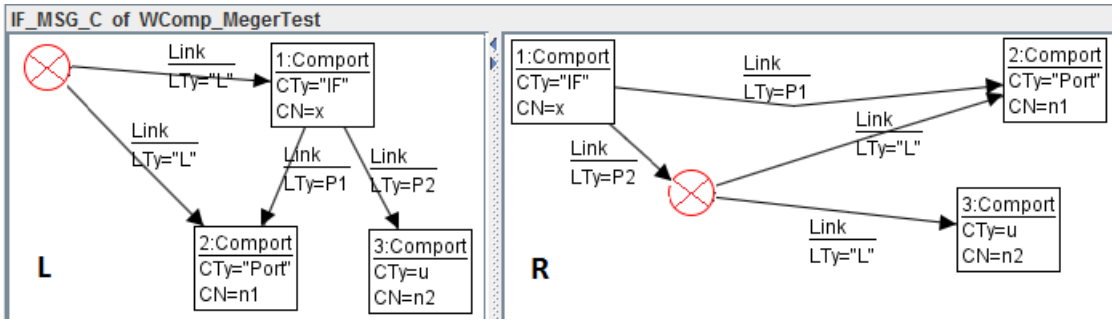


FIGURE A.6 – La Règle de fusion de IF et Port : cas particulier

La règle de la figure A.8 montre que le connecteur de séquence va être aussi dupliqué dans les deux branches d'IF. La logique de cette réécriture est comme suit : quelque soit la valeur de la condition à vérifier par le IF, le comportement qui a été spécifié par la séquence devrait être mis en place d'où la

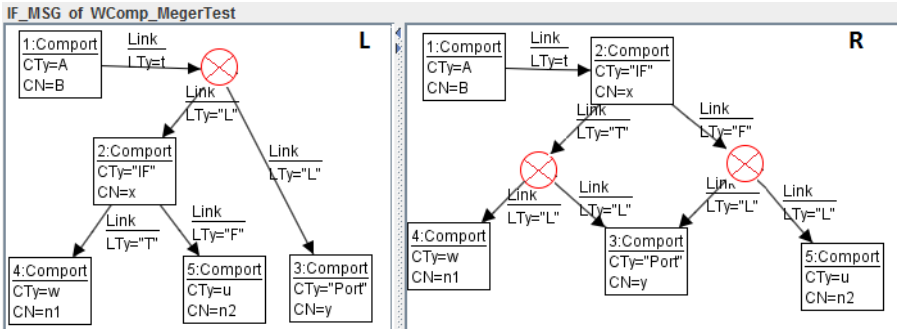


FIGURE A.7 – La Règle de fusion d'IF et Port : cas général

propagation de l'opération de la fusion dans le graphe *R*.

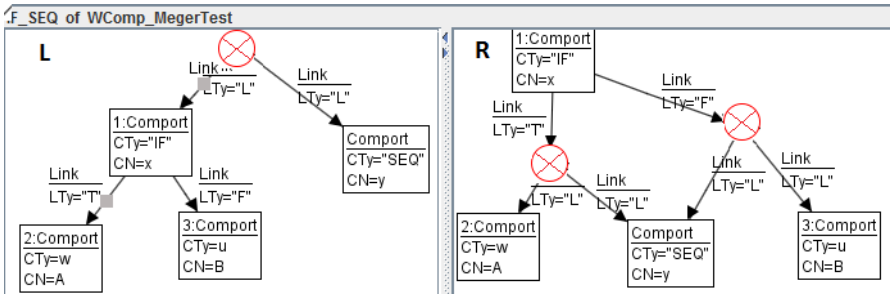


FIGURE A.8 – La Règle de fusion d'IF et SEQ

La réécriture de deux connecteurs *SEQ* La réécriture de deux séquences doit conserver les relations d'ordre établies au sein de chacune séparément. Il existe plusieurs configurations possibles pour deux *SEQ*. Dans la figure A.9 nous représentons le cas où les connecteurs *SEQ* partagent un nœud qui possède le même ordre. Les variables $P1$ et $P2$ remplacent l'une de deux chaîne OUT_1 ou OUT_2 (spécifient l'ordre entre les actions). Si nous supposons que $P1 = OUT_1$ et $P2 = OUT_2$ alors le graphe L de la figure A.9 montre la réécriture de $SEQ(f,h)$ et $SEQ(f,w)$. Intuitivement, la réécriture de ces deux connecteurs implique le résultat donné par le sous-graphe R de cette même figure : la séquence de f et le résultat de fusion de h et w.

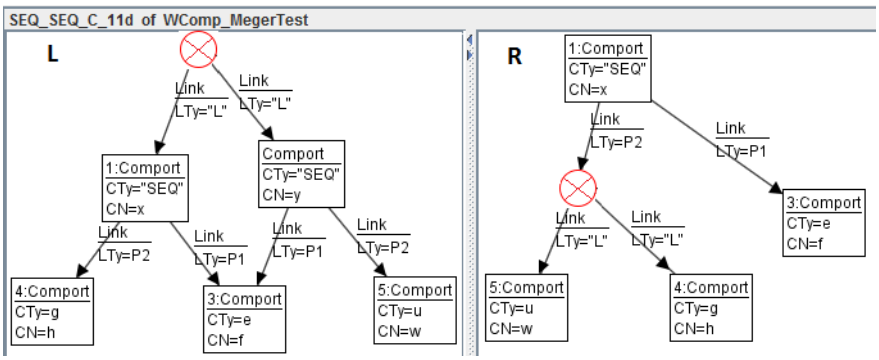


FIGURE A.9 – La Règle de fusion de deux SEQ avec Pivot : cas 1

Une autre configuration possible est lorsque les deux SEQ partagent un nœud qui possède un ordre différent (dans la figure A.10 le nœud h est définit comme un ayant l'ordre OUT_1 dans la première séquence et OUT_2 dans la deuxième séquence), la règle de la figure A.10 sera appliquée. La fusion de $SEQ(h,w)$ et $SEQ(f,h)$ implique comme résultat : $SEQ(f,SEQ(h,w))$.

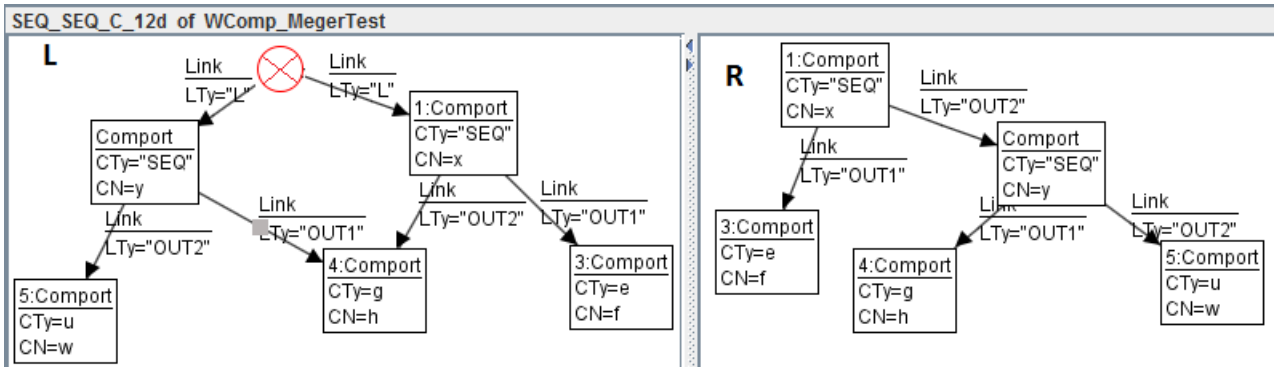


FIGURE A.10 – La Règle de fusion de deux SEQ avec Pivot : cas 2

Dans le cas où les deux séquences partagent un nœud mais ce dernier n'est pas un de leurs successeur immédiate (il existe un chemin de SEQ_1 et un autre chemin de SEQ_2 qui mènent vers ce nœud), les règles des figures A.11 et A.12 seront appliquées. La figure A.11 montre le cas où il existe un nœud en commun entre les branches $P1$ $P3$ et entre les branches $P2$ $P4$.

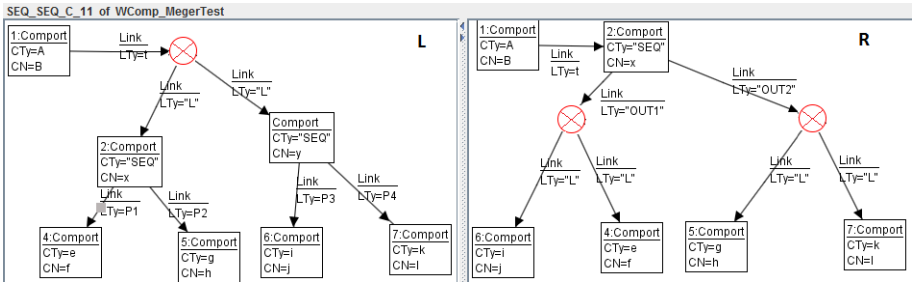


FIGURE A.11 – La Règle de fusion de deux SEQ avec 2 pivots

Un autre résultat possible de pivot entre les deux SEQ est présenté dans la figure A.12. Le nœud h est partagé par les deux branches de la deuxième SEQ qui porte le nom d'instance y (h appartient à la sous-branche OUT_1 et à la fois à la sous-branche OUT_2 de y). La réécriture gardera le nœud f comme le premier nœud a être exécuté et propage l'opération de la réécriture sur la première branche de la deuxième SEQ.

La réécriture de CALL Nous avons déjà mentionné que cet opérateur du langage est utilisé pour la réécriture des liens existants. Les règles de réécriture associée à ce connecteur utilitaire CALL permettent alors de modifier les endroits où la réécriture va être appliquée. La figure A.13 illustre la fusion d'un comportement quelconque x qui possède comme successeur un nœud CALL avec un autre comportement e. Le résultat sera alors la réécriture du lien vers le nœud e pour qu'il soit un successeur immédiat du CALL.

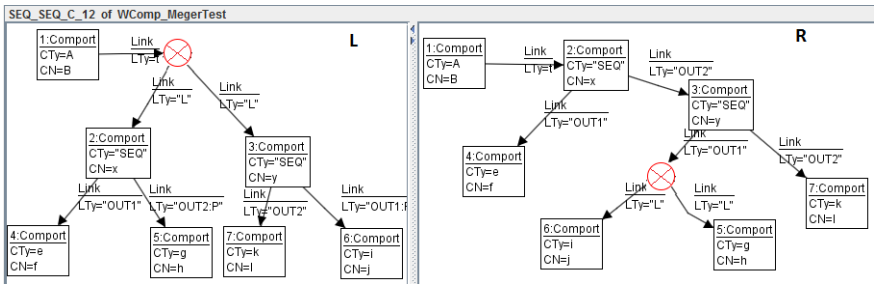


FIGURE A.12 – La Règle de fusion de deux SEQ avec 3 pivots

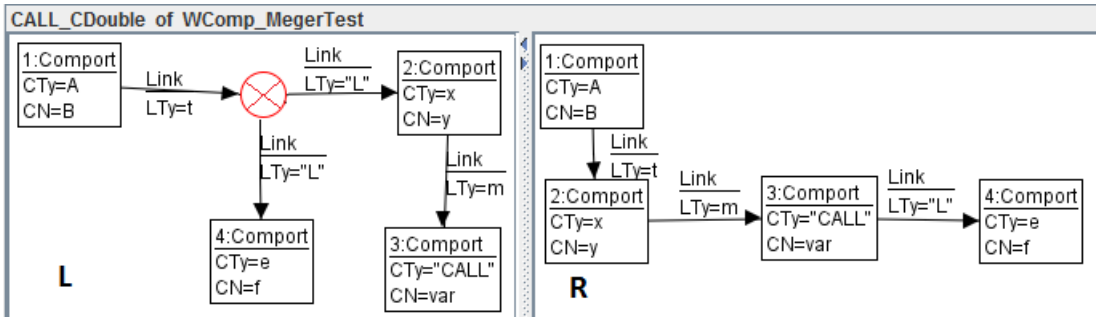


FIGURE A.13 – Règle de fusion de CALL

La réécriture de DELEGATE Cet opérateur est utilisé dans le cas où nous voudrons spécifier qu'un lien doit être unique. C'est-à-dire le nœud qui précède le DELEGATE (dans la figure A.14 c'est le nœud de type A) ne devrait pas avoir d'autres liens vers d'autres nœuds (le lien vers le nœud x sera supprimé).

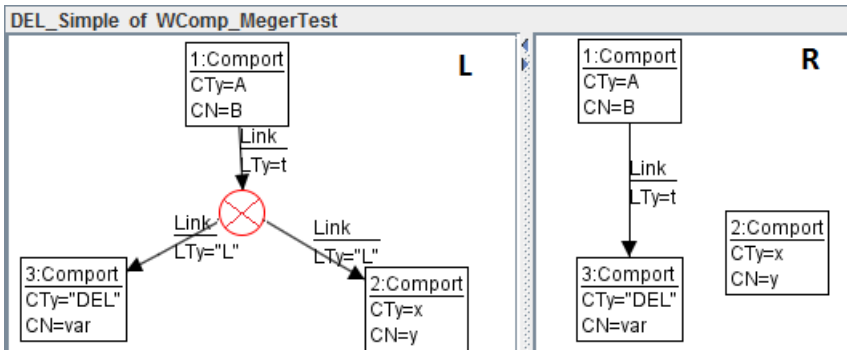


FIGURE A.14 – La Règle de fusion de DELEGATE

La règle de réécriture par défaut Les règles de réécriture spécifiées permettent de fournir, dans le cas où cela est possible, le résultat de fusion des comportements qui sont en interférence. Mais parfois nous aurons des interférences que nous ne pouvons pas résoudre (pas de règle de spécification ni d'automates pour les nouveaux connecteurs). Dans ce cas nous avons prévu une solution par défaut qui consiste à mettre en parallèle les comportements en ajoutant un composant « PAR » en sortie du nœud où l'interférence a été marquée. De ce fait, les comportements en interférence seront exécutés indépendamment les uns des autres. La figure A.15 illustre cette règle de fusion.

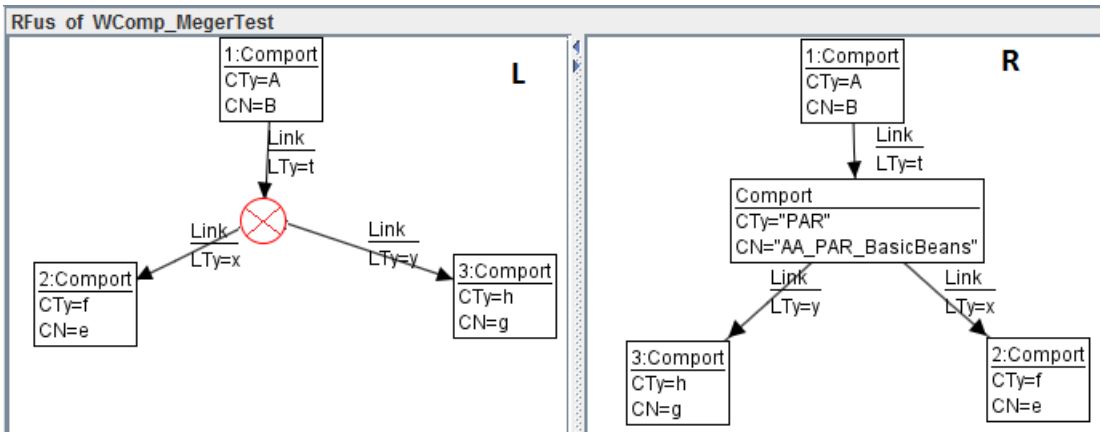


FIGURE A.15 – La Règle de fusion par défaut

A.2 Preuves des propriétés de commutativité, d'associativité et d'idempotence

Pour faire face au caractère imprévisible de l'IAm et pour ne plus se préoccuper de l'ordre d'apparition et disparition des dispositifs (et d'où l'ordre d'application des adaptations), il est important de garantir la propriété de symétrie. Cette propriété devra être maintenu durant toutes les étapes de la composition. Nous avons vu que le processus de composition se déroule en trois étapes : la superposition, la détection et la résolution des interférences. L'opération de superposition effectue l'union des graphes. Elle donc symétrique. La détection des interférences ajoute dans des nœuds le graphe pour marquer les problèmes. Cette opération est aussi symétrique. La dernière opération est la gestion des interférences. Durant cette étape un ensemble de règles de réécriture va être appliqué. Il faudra alors garantir que quel que soit l'ordre dans lequel ces règles vont être appliquées, nous obtenons à la fin le même résultat de résolution (pour un même ensemble d'interférences).

Les propriétés d'idempotence et de commutativité sont obtenues par construction. L'idempotence assure que la réécriture de deux connecteurs qui ont le même ensemble d'entrée produira le connecteur lui même (par exemple $par(a,b) \otimes par(a,b) = par(a,b)$). Nous avons défini une règle générique qui assure l'idempotence pour la réécriture de tous nos connecteurs. La propriété de commutativité est aussi garantie pour nos règles. En effet, la réécriture de deux connecteurs pour une configuration donnée utilise une seule règle. Dans cette règle, il n y a pas un ordre entre les connecteurs. Par exemple, la réécriture de $seq(a,b) \otimes par(a,c)$ applique la même règle que la réécriture de $par(a,c) \otimes seq(a,b)$.

Ce qu'il reste à vérifier c'est la propriété d'associativité. Pour effectuer la preuve de l'associativité, nous avons fait les calculs de toutes les combinaisons possibles entre les connecteurs. Pour ce faire, nous fournissons comme entrée à notre mécanisme de composition un ensemble de graphes représentant les connecteurs de base toute en provoquant un cas d'interférence (les connecteurs ont tous le même nœud prédécesseurs). Le calcul de toutes les combinaisons possibles permet de détecter les règles de réécriture où la propriété de symétrie a échoué. Dans un tel cas le super-designer est prévenu pour intervenir et modifier les règles en question avant de refaire de nouveau la preuve. Cette preuve ne peut pas être faite manuellement vu le nombre de règles à vérifier et les combinaisons possibles de connecteurs.

Nous avons défini un outil pour faciliter la preuve de l'associativité. La preuve est faite à chaque fois qu'il y a un ajout manuel d'un nouveau connecteur. Pour ce faire, nous avons besoin en entrée

du sous-graphe du nouveau connecteur ainsi que de ces règles de réécriture avec le reste des connecteurs (la preuve a été déjà faite pour cet ensemble de connecteurs). Cette preuve consiste à calculer la réécriture de toutes les configurations possibles pour toutes les combinaisons possibles de 3 connecteurs (le nouveau avec deux anciens). Nous avons créé une interface facilitant la génération de ces sous-graphes. La figure A.16 illustre l'interface utilisée pour atteindre cet objectif. Il faudra spécifier le nombre de variables nécessaires pour nos connecteurs. Par défaut, nos connecteurs sont binaires. Ensuite il faudra spécifier les noms de variables qui seront utilisés dans la définition des connecteurs. Finalement il faudra choisir à travers des listes de sélections pour quels connecteurs nous souhaitons faire la preuve.

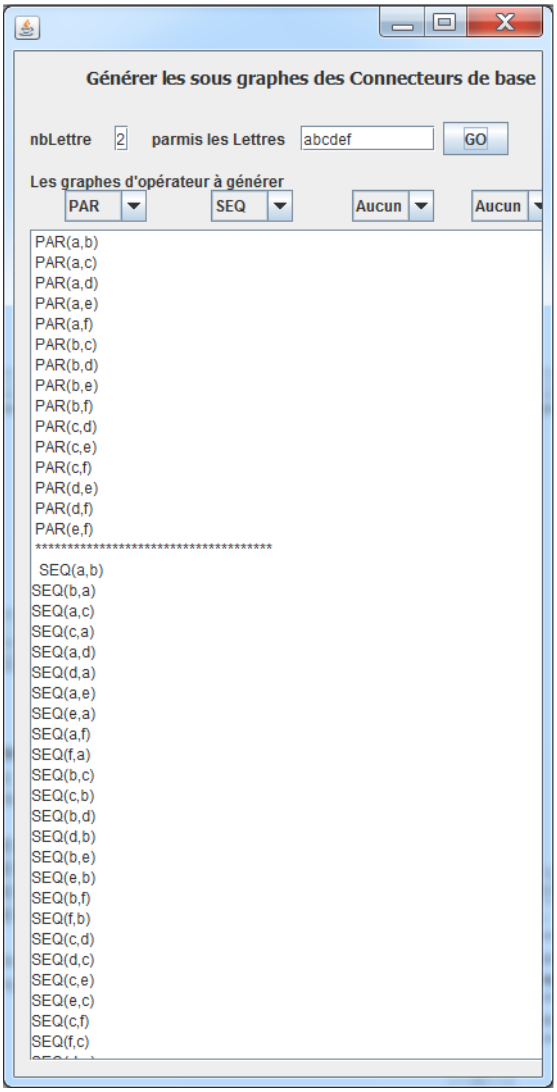


FIGURE A.16 – Générer automatiquement les sous-graphes d'entrée pour la preuve

Une fois que le jeu d'entrées a été généré, le preuve automatique se déroule en utilisant le graphe du nouveau connecteur et les règles de réécriture. Les champs de texte False (respectivement True) affichera le nombre de cas où le test a échoué (réussi). Les cas d'échec seront détaillés dans le champs en bas de la fenêtre. L'interface est illustrée dans la figure A.17

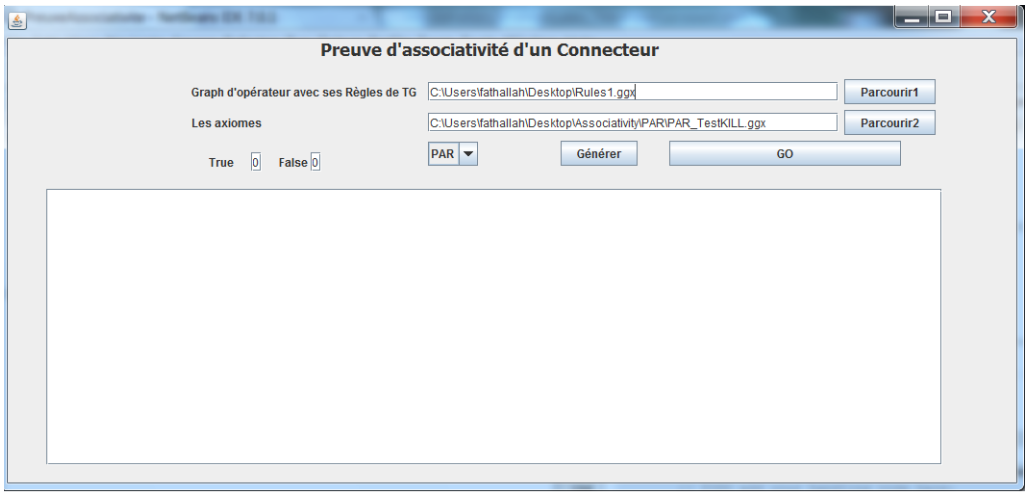


FIGURE A.17 – L'écran d'accueil de l'outil de preuve de l'associativité

A.3 Evaluation de la gestion des interférences

L'étude des temps de réponse de notre mécanisme d'adaptation a été présenté dans [TLR+12]. L'approche de résolution que nous proposons dans cette thèse s'appuie sur une autre technique qui est la réécriture de graphe. Par conséquent, le temps de réponse va être différent de celui qui a été évalué. Nous présentons dans cette section en détail les temps de réponse de l'opération principale de la composition qui est la gestion des interférences. Nous évaluons notre approche en termes de performances liées aux expériences de compositions d'assemblages des composants générés aléatoirement. Ils ont été réalisés sur un ordinateur personnel standard (Intel Core 2, 3.06 GHz).

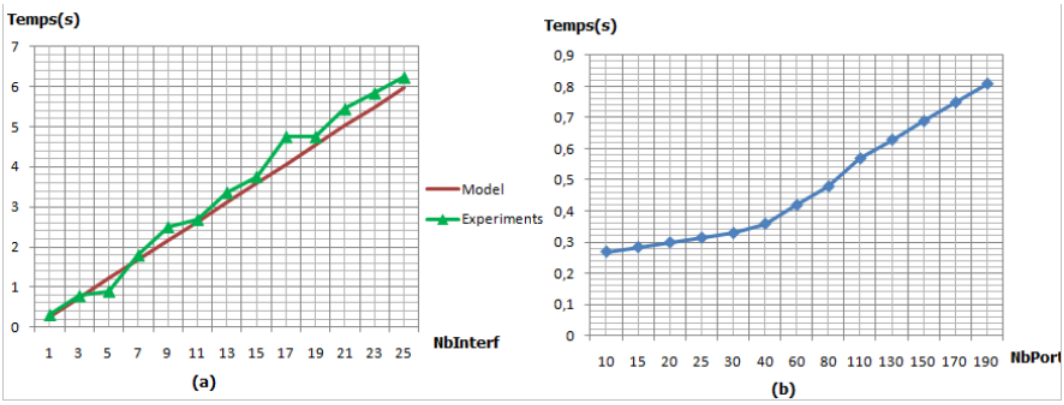


FIGURE A.18 – Temps de réponse du mécanisme de résolution des interférences

La complexité de l'implémentation actuelle est fortement liée au moteur de transformation de graphe utilisé AGG Attributed Graph Grammar [Tae00] car la majorité du temps de composition est passée dans la phase de résolution des interférences. L'opération la plus complexe durant la phase d'application d'une règle de transformation de graphes est la recherche d'un motif dans le graphe à modifier (rechercher s'il existe le sous-graphe L de la règle de transformation) c'est de l'ordre de $O(2^{nnode})$ avec *NNode* est le nombre de nœuds dans la partie gauche L de la règle. Si nous supposons que nous exécutons une règles de transformation pour chaque cas d'interférence le coût total d'exé-

cution du moteur de transformation de graphe est : $CoutMTG = a_1 * NbInterf * + a_2$; avec nk est le nombre de nœuds de la règle R_k, $NbInterf$ est le nombre d'interférences à résoudre et a1 et a2 sont les paramètres du modèle. Dans nos règles n_k est entre 3 et 13. La moyenne est $n_k = 8$.

Le temps passé par AGG pour exécuter une règle de fusion varie entre $0,052$ et $0,366$ seconde. Ceci est lié au nombre de nœuds dans une règle. Il faut en moyenne 0,248 s pour exécuter une règle. La figure A.18.a montre le temps de réponse du moteur de transformation fonction du nombre d'interférences. Selon ces expériences $a_2 = 0,02$ et $a_1 = 0,446$. Le nombre de composants présent dans l'application n'a pas d'influence sur le temps de réponse de la phase de résolution des interférences. Il affecte le temps de réponse global du mécanisme de la fusion (utilisée pour calculer le pivot et l'export du graphe vers la plate-forme), Dans la figure A.18.b, nous montrons l'influence du nombre de composants sur le temps de réponse total du moteur de fusion. Nous avons fait varier le nombre de composants instanciés (NbPort) dans l'application de 10 à 190. Dans cette expérimentation, nous avons utilisé un graphe d'application avec deux points d'interférences à résoudre.

ANNEXE B

Le détail de la génération des règles de réécriture

Cette annexe s'intéresse à la méthodologie de génération des règles de réécriture. La première partie est consacrée à la définition des règles pour l'ensemble de nos connecteurs de base. La deuxième partie présente la génération des règles pour des nouveaux connecteurs en exploitant leur description comportementale.

B.1 Règles de réécriture des connecteurs de base

La méthodologie de génération des règles de réécriture est donnée pour les connecteurs de déclenchement de type $n-1$. Nous présentons le détail de génération des règles de réécriture pour les connecteurs *AND* et *OR* en déroulant toutes les configurations possibles.

B.1.1 Règles pour la réécriture du connecteur AND

Pour rappel, le connecteur *AND* est utilisé pour synchroniser l'appel au port de sortie qui est conditionner par la présence de deux événements d'entrée. Dans la suite, nous utilisons les variables$\{a,b,c,d\}$ pour désigner les événements d'entrée des connecteurs et la variable Out_1 pour la sortie. Pour simplifier, nous présentons directement la table de vérité du connecteur extraite à partir de l'automate sans illustrer ce dernier.

B.1.1.1 Réécriture de deux connecteurs AND

Deux connecteurs *AND* peuvent avoir plusieurs configurations possibles selon le nombre d'événements d'entrée qu'ils partagent. Il est possible qu'ils partagent un événement, deux événements ou bien aucun.

cas 1 : $out_1 = AND(a,b)$ $out_1 = AND(b,c)$ Nous présentons le cas où les deux connecteurs *AND* ont en commun un événement d'entrée b. Les tables de vérité de chacun de deux connecteurs sont comme suit :

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

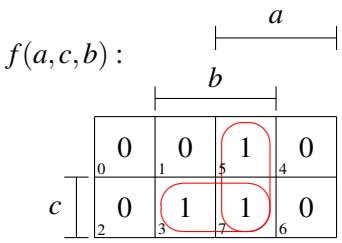
b	c	f
0	0	0
0	1	0
1	0	0
1	1	1

La composition des comportements de ces deux connecteurs peut se faire selon diverses stratégies. Dans la suite nous utilisons la composition parallèle. Ceci signifie que l'activation de l'évènement

de sortie par l'un de nos connecteurs déclenche l'appel du port de sortie là où l'interférence a été identifiée.

a	b	c	$f(a,b,c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

La table de vérité résultant de cette opération de composition va être exploitée pour déterminer l'équation booléenne de la sortie Out_1. Pour avoir une équation simplifiée nous passons par le tableau de Karnaugh qui est le suivant :



La simplification proposée par la table de Karnaugh consiste à faire les groupements des 1 sous-jacents (selon un ensemble de règles à respecter). Nous obtenons l'équation suivante :

$$Out1 = b \cdot (a + c)$$

Cette équation sera par la suite traduite vers une règle de réécriture de graphes. Cette règle est illustrée par la figure B.1

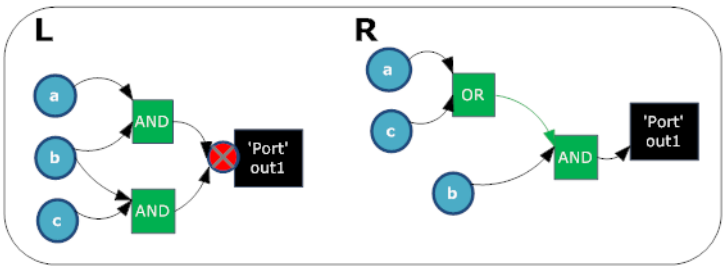


FIGURE B.1 – Règle de réécriture pour : $out_1 = AND(a,b)$ $out_1 = AND(b,c)$

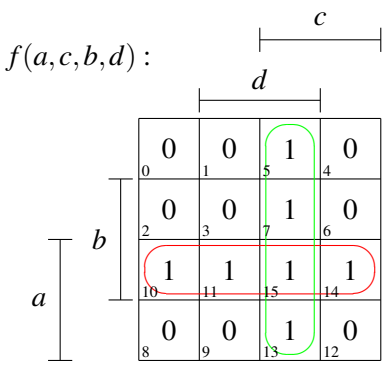
cas 2 : $AND(a,b) \otimes AND(c,d)$

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

c	d	f
0	0	0
0	1	0
1	0	0
1	1	1

a	b	c	d	$f(a,b,c,d)$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

La table de vérité résultant de cette opération de composition va être exploitée pour déterminer l'équation booléenne de la sortie Out_1. Pour avoir une équation simplifiée, nous passons par le tableau de Karnaugh qui est le suivant :



La simplification proposée par la table de Karnaugh consiste à faire les groupement des 1 sous-jacents (selon un ensemble de règles à respecter). Nous obtenons l'équation suivante :

$$Out1 = a \cdot b + c \cdot d$$

Cette équation sera par la suite traduite vers une règle de réécriture de graphes. Cette règle est illustrée par la figure B.2.

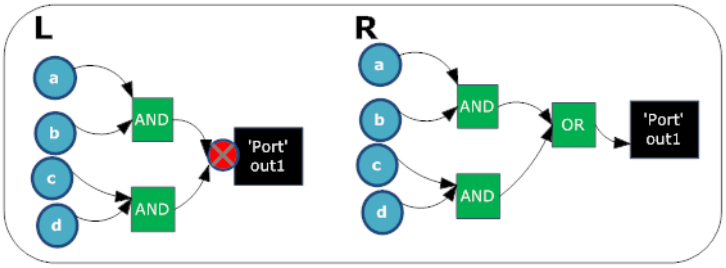


FIGURE B.2 – Règle de réécriture pour : $AND(a,b) \otimes AND(c,d)$

cas 3 : $AND(a,b) \otimes AND(a,b)$ Ce cas d'interférence se produit dans le cas où au moins deux adaptations utilisent un même type de connecteur avec les mêmes évènements d'entrée. Nous avons vu que nous garantissons l'idempotence par construction. Le résultat attendu dans ce cas d'interférence est de ne garder que l'un de deux connecteurs avec ces événements d'entrée. La table de vérité résultant de cette composition de connecteur est la suivante :

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

Il s'agit de la table de vérité d'un connecteur *AND*. L'équation est donc celle du connecteur *AND* :

$$Out1 = a \cdot b$$

La règle de réécriture est présentée par la figure B.3.

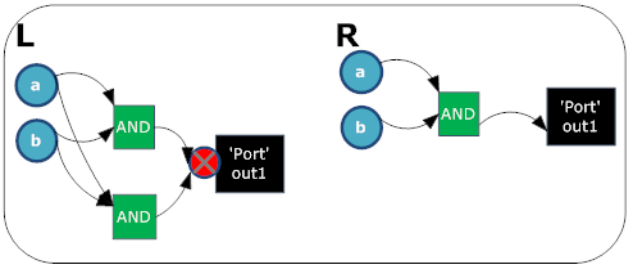


FIGURE B.3 – Règle de réécriture pour : $AND(a,b) \otimes AND(a,b)$

B.1.1.2 Réécriture des connecteurs AND et OR

Comme dans le cas de la réécriture de deux connecteurs *AND*, la réécriture d'un connecteur de type *AND* et un connecteur de type *OR* est régi par plusieurs règles dont chacune correspond à une configuration particulière.

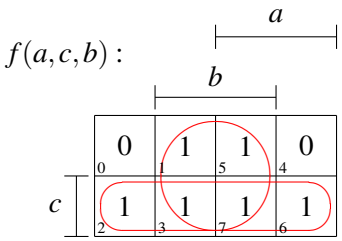
cas1 : $AND(a,b) \otimes OR(b,c)$ Les deux connecteurs *AND* et *OR* peuvent partager un évènement d'entrée qui est b dans ce cas. Les tables de vérités de ces connecteurs ainsi que celle résultant de leur composition sont les suivantes :

a	b	$f = AND(a,b)$
0	0	0
0	1	0
1	0	0
1	1	1

b	c	$f = OR(b,c)$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	c	$out1$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Le tableau de Karnaugh est le suivant :



L'équation booléenne obtenue à partir du tableau du Karnaugh est comme suit :

$$Out1 = c + b$$

La réécriture de ces deux connecteurs va garder le comportement le moins restrictif puisque nous avons appliqué une composition parallèle. La présence de l'évènement b déclenchera l'appel du port de sortie $out1$ quelque soit l'état de l'événement d'entrée a. La règle de réécriture est présentée par la figure B.4.

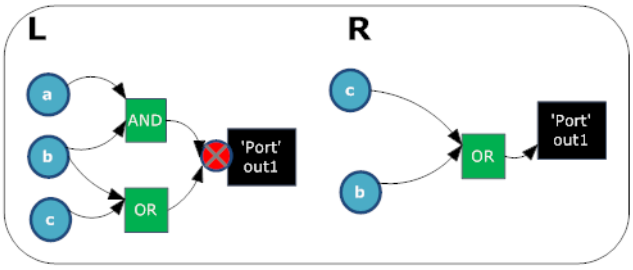
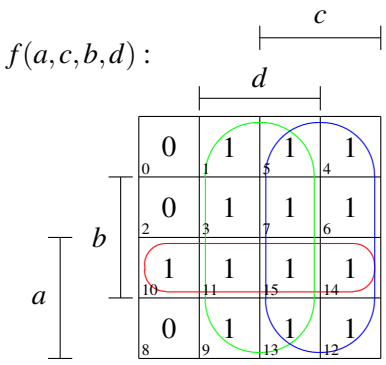


FIGURE B.4 – Règle de réécriture pour : $AND(a,b) \otimes OR(b,c)$

cas2 : $AND(a,b) \otimes OR(c,d)$ Nous avons déjà présenté les tables de vérité des connecteurs *AND* et *OR*. La table de vérité de leur composition selon la configuration de ce cas est la suivante :

a	b	c	d	$f(a,b,c,d)$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

La table de vérité résultant de cette opération de composition va être exploitée pour déterminer l'équation booléenne de la sortie Out_1. Pour avoir une équation simplifiée, nous passons par le tableau de Karnaugh qui est le suivant :



La simplification proposée par la table de Karnaugh consiste à faire les groupements des 1 sous-jacents (selon un ensemble de règles à respecter). Nous obtenons l'équation suivante :

$$Out1 = a \cdot b + c + d$$

La traduction de cette équation vers une règle de réécriture est illustrée par la figure B.5.

cas3 : $AND(a,b) \otimes OR(a,b)$ En suivant la même démarche selon une composition parallèle, la composition de ces deux connecteurs produira l'équation booléenne suivante :

$$Out1 = a + b$$

La règle de réécriture est donnée par la figure B.6.

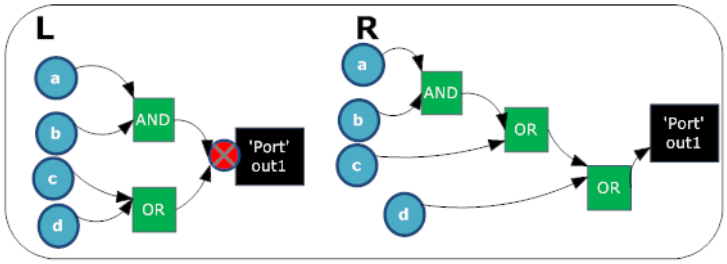


FIGURE B.5 – Règle de réécriture pour : $AND(a,b) \otimes OR(c,d)$

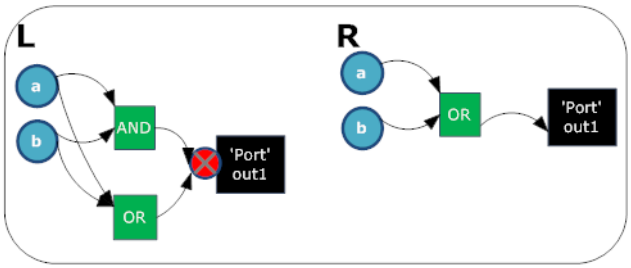


FIGURE B.6 – Règle de réécriture pour : $AND(a,b) \otimes OR(a,b)$

B.1.1.3 Réécriture des connecteurs AND et l'envoi d'un message

Il existe deux configurations possibles pour un connecteur *AND* et un envoi de message (un port) : (1) le port appartient à l'ensemble d'événements d'entrée du connecteur *AND* et (2) il s'agit d'un port différent.

cas 1 : $AND(a,b) \otimes b$ La composition que nous utilisons dans cette annexe suit une logique permissive. Dans ce cas, le déclenchement du port de sortie où l'interférence a été détectée dépendra principalement de la présence du port b, c'est à dire quelque soit l'état de l'évènement d'entrée a dès l'activation de la sortie s'effectue sur la présence du b. L'équation déterminant le déclenchement du port de sortie est comme suit :

$$Out1 = b$$

La traduction de cette équation vers une règle de réécriture donne la figure B.7.

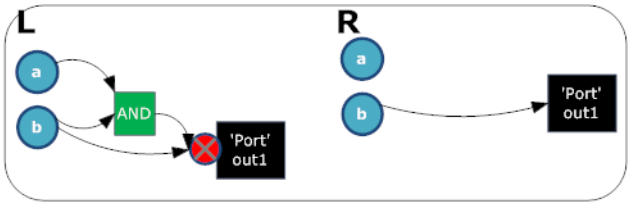


FIGURE B.7 – Règle de réécriture pour : $AND(a,b) \otimes b$

cas 2 : $AND(a,b) \otimes c$ Lorsque le message ne fait pas partie de l'ensemble d'événements d'entrée du connecteurs *AND* la composition ajoute un connecteur *OR* (c'est une composition parallèle). L'équation booléenne relative à cette composition est la suivante :

$$Out1 = a \cdot b + c$$

La règle de réécriture relative à cette configuration est illustrée dans la figure B.8.

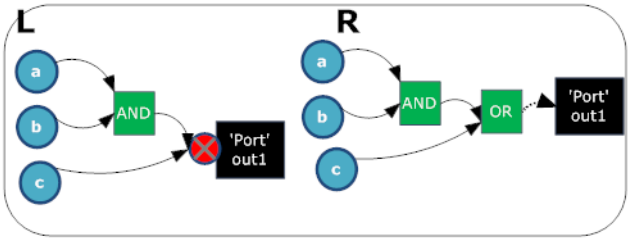


FIGURE B.8 – Règle de réécriture pour : $AND(a,b) \otimes c$

B.1.2 Règles pour la réécriture du connecteur OR

B.1.2.1 Réécriture de deux connecteurs OR

Plusieurs configurations sont possibles entre deux connecteurs *OR*. Dans cette section nous présentons une seule configuration : les deux connecteurs partagent un seul événement d'entrée.

cas 1 : $OR(a,b) \otimes OR(b,c)$ La table de vérité correspondant à la composition de ces deux connecteurs selon cette configuration est comme suit :

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

L'équation obtenue suite à la simplification à travers la table de Karnaugh est la suivante :

$$Out1 = a + b + c$$

La transformation de l'équation vers une règle de réécriture donne le résultat présenté par la figure B.9.

B.1.2.2 Réécriture des connecteurs OR et l'envoie d'un message

cas1 : $OR(a,b) \otimes b$ La composition du connecteur *OR* avec un envoi de message appartenant à l'ensemble de ces événements d'entrée donne comme résultat l'équation booléenne suivante :

$$Out1 = a + b$$

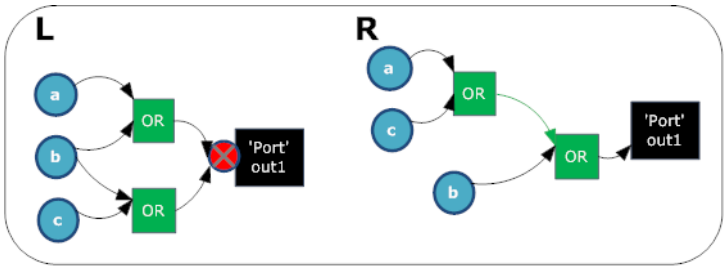


FIGURE B.9 – Règle de réécriture pour : $OR(a,b) \otimes OR(b,c)$

Jusqu'à présent nous avons détaillé la démarche pour la définition des règles de réécriture pour les connecteurs de base. Cette méthodologie sera utilisée pour l'extension de l'ensemble de connecteurs en rajoutant des nouveaux connecteurs. Ces connecteurs fournissent une description de leur comportement sous forme d'automate.

B.2 L'ajout d'un nouveau connecteur X

Nous supposons qu'un nouveau connecteur X a été ajouté pendant l'exécution. Ce connecteur apparaît avec la description de son comportement à l'aide d'un automate. Le tableau de vérité (obtenue à partir de l'automate) de ce connecteur détaille la valeur de sa sortie en fonction de ces entrées et il est comme suit :

a	b	f
0	0	1
0	1	0
1	0	0
1	1	1

L'équation booléenne de ce connecteur est la suivante :

$$S = \overline{a} \cdot \overline{b} + a \cdot b$$

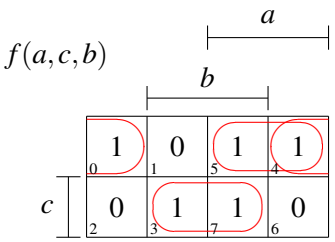
Pour que le nouveau connecteur soit traité de la même manière que les connecteurs de base, il est indispensable d'ajouter des règles de réécriture pour ce connecteur avec tous les autres. Dans la suite, nous présentons le détail de génération des règles pour le connecteur X.

B.2.1 Réécriture des X et AND

$AND(a,b) \otimes X(b,c)$ Le premier cas que nous détaillons correspond à celui où les deux connecteurs partageant un seul évènement d'entrée. La table de vérité obtenu à l'issue de la composition est la suivante :

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

La représentation de cette table de vérité sous forme du tableau de Karnaugh est comme suit :



L'exploitation de cette table produira l'équation booléenne suivante :

$$S = b \cdot c + \overline{c} \cdot (a + b)$$

Cela correspond à la règle de réécriture de la figure B.10.

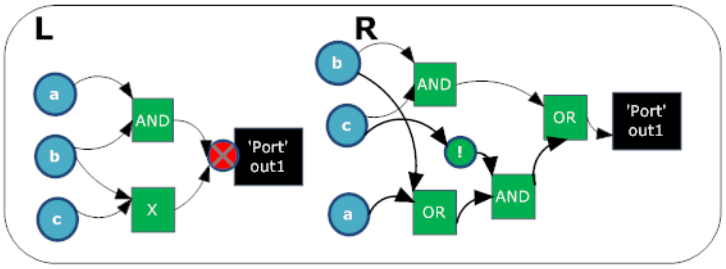


FIGURE B.10 – Règle de réécriture pour : $AND(a,b) \otimes X(b,c)$

$AND(a,b) \otimes X(a,b)$ Les connecteurs *AND* et *OR* peuvent partager le même ensemble d'évènements d'entrée. Dans ce cas, la table de vérité obtenue à l'issue de leur composition se définit comme suit :

a	b	f
0	0	1
0	1	0
1	0	0
1	1	1

Nous remarquons que cette table de vérité est la même que celle du connecteur X. L'équation booléenne sera alors celle de ce connecteur (exprimé à l'aide des connecteurs de base). La règle de réécriture est illustrée par la figure B.11.

$$Out1 = \overline{a} \cdot \overline{b} + a \cdot b$$

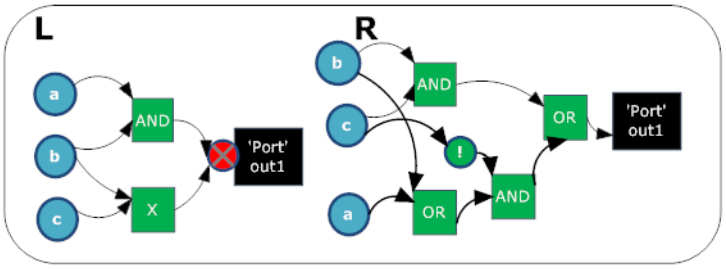


FIGURE B.11 – Règle de réécriture pour : $AND(a,b) \otimes X(a,b)$

B.2.2 Réécriture des X et OR

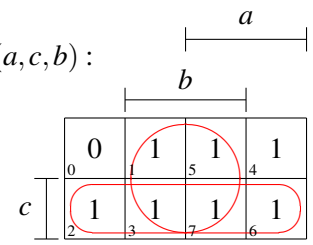
Le nouveau connecteur X pourra être réécrit avec le connecteur OR. Nous détaillons les mêmes configurations que celles qui ont été faites pour le connecteur AND.

$OR(a,b) \otimes X(b,c)$ La composition des connecteurs OR et X dans le cas où ils partagent un événement d'entrée produit la table de vérité suivante :

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Le tableau de Karnaugh est le suivant :

$f(a,c,b)$:



L'équation booléenne obtenue suite à la composition de ces opérateurs est comme suit :

$$Out1 = c + a + b$$

$OR(a,b) \otimes X(a,b)$ La composition d'un connecteur X avec un connecteur OR avec un partage complet de l'ensemble d'évènements d'entrée donne comme résultat l'équation booléenne suivante :

$$Out1 = a + b$$

B.2.3 Réécriture des X et un envoie de message p

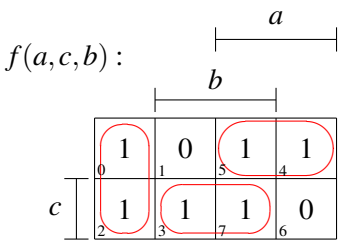
$$S = \overline{a} + b$$

B.2.4 Réécriture du connecteur X avec lui même

$X(a,b)\ X(b,c)$ Le nouveau connecteur pourra être utilisé par plusieurs adaptations. De ce fait, une interférence pourra être détectée entre les nouveaux connecteurs. Dans le cas où ces connecteurs partagent un événement d'entrée, la table de vérité se présente comme suit :

a	b	c	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Le tableau de Karnaugh représentant cette table de vérité est le suivant :



La simplification faite à travers le tableau de Karnaugh donne l'équation suivante

$$Out1 = \overline{a} \cdot \overline{b} + c \cdot b + \overline{c} \cdot a$$

Table des figures

Liste des tableaux